

Bachelor Thesis

**Visual Authoring of
CI/CD Pipeline Configurations**

Sebastian Teumert

<https://orcid.org/0000-0002-6483-3162>

March 31, 2021

Reviewer:

Prof. Dr. Bernhard Steffen

Tim Tegeler

TU Dortmund University

Department of Computer Science

Chair for Programming Systems (Chair V)

<http://ls5-www.cs.tu-dortmund.de>

Acknowledgements

"What A Long, Strange Trip It's Been ..." — The Grateful Dead

I would like to thank the following people, who supported writing this thesis during a global pandemic, for their patience, guidance and support:

- Tim Tegeler; for his great advice, timely and accurate responses and mentorship.
- My brother; for preventing the most glaring typos and grammatical errors this thesis might otherwise have suffered from.
- My friends (they know who); for sharing my joy of finally being able to write a thesis and finishing my studies.
- My parents; for their continued support of my studies. It has become a long journey, but it will finally bear fruit, and I am very grateful for the support I have received and hope to continue to receive for my masters degree.

All your contributions are greatly appreciated.

Contents

1	Introduction	1
2	Related Work	5
2.1	3D Modelling	6
2.1.1	Blender - Shader Nodes	6
2.1.2	Unreal Engine - Blueprint Editor	6
2.2	Other Fields	8
2.2.1	Data Transformation: Azure Data Factory	8
2.3	CI/CD Solutions	9
2.3.1	SemaphoreCI	9
2.3.2	GitLab: Pipeline Editor	10
2.4	A Model-driven Approach	11
2.5	Important Takeaways	13
3	A Graphical DSL for Visual Authoring	15
3.1	CINCO - A Meta-modelling Framework	15
3.2	Preconsiderations	16
3.3	Approaching Polymorphic Values	17
3.4	The Elements of the DSL	18
3.5	Color Scheme & Iconography	21
3.5.1	Color Scheme	21
3.5.2	Iconography	22
4	Deriving Configurations from the Graph Model	25

5	Results	31
5.1	Case Study: A Modern, Cloud-Based Web Application	31
5.2	Summary	33
6	Future Work	35
6.1	Ensuring Instantiation of Prerequisite Jobs	35
6.2	Generalization	39
6.3	Visual Authoring for Regular Expressions	44
6.4	Reducing Edge Count via Target Extension	45
6.5	Towards Reusability	47
6.5.1	Prime References	47
6.5.2	Abstract Targets	47
6.5.3	Sharing Definitions via Marketplace	48
6.6	Rig as Integrated Development Environment (IDE)	48
6.7	Bootstrapping Rig	50
7	Discussion	53
7.1	Edge Proliferation	53
7.2	Comparison with SemaphoreCI	54
7.3	Comparison with GitLab's Visual Pipeline Editor	56
7.4	CINCO as Workbench	56
8	Conclusion	59
A	Further Information	61
	Table of Figures	64
	Bibliography	69

Chapter 1

Introduction

Continuous practices have become increasingly popular in the last two decades and are now a staple of modern software development. Starting with *continuous integration*, which was popularized by Martin Fowler in 2000 [19] and revised in 2006 [18], continuous practices have evolved to also include *continuous deployment* and *continuous delivery*. As an encompassing field, *Continuous Software Engineering (CSE)* has become increasingly popular within the wider DevOps movement.

A representation of the so called *DevOps lifecycle* is shown in Figure 1.1. DevOps aims to bridge the gap between developers and the operations team so that software can be build,

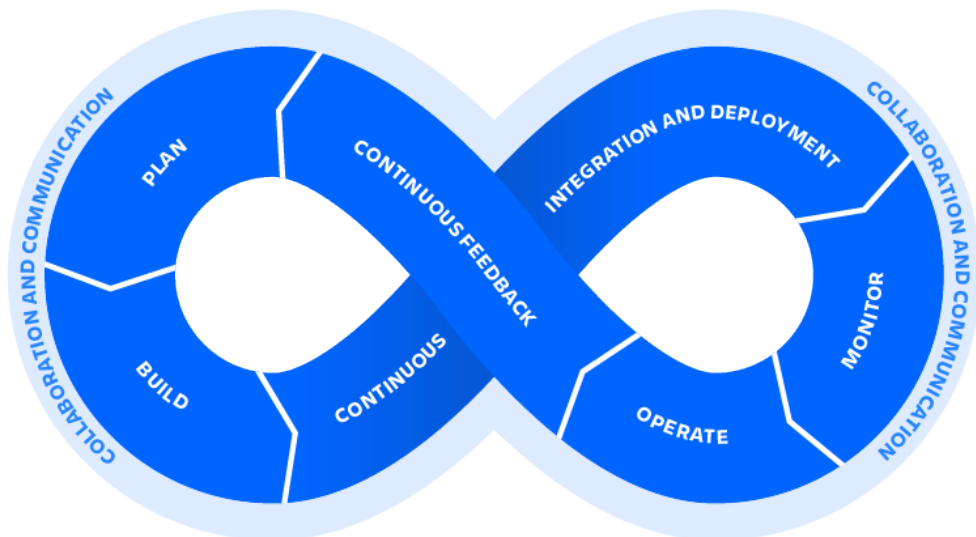


Figure 1.1: DevOps Infinity Wheel, as shown in [2].

tested and released (or deployed) faster [2]. The enabling, and thus central element of this process is *Continuous Software Integration and Deployment* (CI/CD).

CI/CD workflows are executed in so called CI/CD pipelines. While such pipelines offer great value to teams not adopting the DevOps process by improving the developer experience through quick feedback on their changes, they are instrumental for allowing teams to gradually adopt the full DevOps lifecycle and mindset. Without the ability to quickly build, test and integrate each software version, the loop as shown in Figure 1.1 becomes impossible to perform. As such, one of the most important steps to start the conversion towards a DevOps mindset is implementing a CI/CD pipeline.

Thus, CI/CD is becoming increasingly popular and important in modern software engineering, especially as one of the six phases of the *DevOps lifecycle* [2]. With the increasing adoption of the DevOps methodology and mindset, increasing pressure is put on developers to support CI/CD pipelines in their projects and to be able to read and write correct configurations for such pipelines.

Since CI/CD is concerned with building, integrating and deploying a specific *version* of the software, it is intrinsically linked to the concept of *version control systems* (VCS) (e.g. SVN, Mercurial, Git, Bazaar). The configuration for the CI/CD pipeline is usually stored inside the repository (cf. Table 1.1) itself. A CI/CD pipeline can be triggered by a variety of events, but by far the most common triggers are new commits (a revision in SVN, a commit hash in Git) or new tags, both of whom indicate the presence of a new version of the software being available, which needs to be tested, integrated and build (and optionally, deployed). Other *events* (which can be configured depending on the platform) may also trigger a new pipeline run, e.g. scheduled pipelines for nightly builds, or builds based on external API calls.

A review of existing CI/CD solutions suggests that YAML has become exceedingly popular as a language for the configuration files of CI/CD pipelines (cf. Table 1.1). Some of these platforms offer visualizations with varied, but usually limited, editing and authoring capabilities for such workflows. A look outside of CI/CD pipelines reveals that YAML is also used for similar configurations, e.g. for Azure Data Factory Pipelines [13].

Three of the most popular source code hosting platforms (GitLab, GitHub and BitBucket) now also offer CI/CD pipelines which can be configured with YAML files. A repository may include the necessary configuration files, and the CI/CD pipeline is then run on isolated runners provided by the platform or by specialized runners provided by the user [26] [23] [21].

Platform	Path(s)	Visualization	Visual Authoring
GitLab	<code>.gitlab-ci.yml</code>	Partial ¹	No ²
GitHub	<code>.github/*.yml</code>	No	No
BitBucket	<code>bitbucket-pipeline.yml</code>	No	No
Travis	<code>.travis.yml</code> [47]	No	No
CircleCI	<code>.circleci/config.yml</code> [12]	No	No
AppVeyor	<code>appveyor.yml</code> [1]	No	No ³
SemaphoreCI	<code>.semaphore/semaphore.yml</code>	Partial ⁴	Limited ⁵
Azure DevOps	<code>azure-pipelines.yml</code>	Rudimentary ⁶	No ³

Table 1.1: CI/CD solutions and how they can be configured

Especially in smaller projects, developers who wish to implement or adopt the DevOps lifecycle find themselves tasked with configuring the complex workflow of such a pipeline, a task that requires them to become familiar with a whole range of new concepts, from the syntax of the configuration file to the used keywords and their semantics. But a more easily accessible tool for the configuration of such pipelines not only benefits developers, but also operations teams.

While YAML was designed as a *"human friendly data serialization standard"* [4], writing correct pipeline configurations remains an error-prone task. Many of the problems presented in the following paragraphs have been previously laid out in [46].

- Despite the ability to describe data formats written in JSON (and thus subsequently YAML) via schema information [53], support for this has largely been unimplemented by vendors of CI/CD solutions (Azure DevOps *does* offer an API that returns a schema [36], but this is an exception, not the norm). This means the author can not leverage any support of their IDE or authoring tool wrt. providing correct values. Some platforms, e.g. GitLab, offer a linting tool [30] to combat this, but the lack of code-assist remains and the requirement to copy code back and forth between the tool and the editor of the authors choice is an additional complication.
- Re-usability of configurations seems limited. Copy & paste is often an accepted form of writing new configurations. Although many platforms offer templates to get users started, it is often required to combine multiple templates when configuring more complex workflows.

¹ Only jobs and their order, either as stages (see Figure A.3) or DAG (see Figure A.2)

² The *visual editor* [31] does not allow interaction. Writing YAML is still required.

³ UI configuration is possible to some extent.

⁴ Only jobs, task and the order of execution

⁵ Some features can be configured via the UI, interaction with the visualization is limited.

⁶ Only jobs and their order; for pipelines that have already been instantiated.

- Discoverability of features is low. It is often necessary for users to peruse lengthy documentation files to even start with basic workflows [32] [31].
- In case of GitLab, the assignment of jobs to build stages is not straight-forward, especially when adding new jobs mid-process later on. The dependencies between jobs easily become non-obvious when looking at a large text file, although efforts have been undertaken by some platforms to visualize these dependencies as shown in Table 1.1.
- The process of creating correct configuration files is thus convoluted and error-prone. It often involves trial & error and uploading new versions of the configuration files to the actual repository, waiting to see if the pipeline works as expected. This also pollutes the commit history with meaningless commits unless the history is rewritten retroactively, which is inadvisable in public repositories from which someone else might have pulled or cloned in the interim. Sometimes this can be avoided by forking the repository and experimenting in a separate copy, but this is neither desirable nor always feasible.

To address these shortcomings, a model-driven approach to *Continuous Practices* has been suggested in [46]. Therein, the authors suggest a model-driven approach which focuses on targets that can be used to parameterize jobs and a graphical DSL to model jobs as graph.

This thesis expands on that approach and develops a DSL with CINCO, a graphical DSL meta-modelling tool [9]. The goal is to produce a visual authoring tool for CI/CD pipelines, which only allows for correct models to be built. By using a graphical DSL, the author of a workflow can visually see how jobs flow into each other, can see the dependencies and how the jobs are configured, and can concentrate on creating a logically sound build process, instead of having to tinker with the intricacies of YAML and the target platform.

To this end, the DSL is not only described syntactically (Chapter 3), but also semantically, so that a correct interpretation can be derived (Chapter 4). Some reasonable alternatives to the presented interpretation are also given, as well as the rule-based theoretical background on which the code generator was based. Although the focus of this work was GitLab as a target platform, possible generalizations of the problem have been investigated and are briefly discussed as opportunities for future work in Chapter 6.

The visual authoring tool using the DSL has been dubbed "Rig". CI/CD pipeline configurations form an intricate web of connections; connecting jobs, targets, properties and other DSL elements via edges. The rig is the arrangement of sails and the rigging - the system of ropes, chains and cables holding the sails to the mast [52]. Just like a ship must be rigged before sailing, the CI/CD pipeline must be rigged before the software can be shipped continuously.

Chapter 2

Related Work

Before proposing a solution, this thesis reviews some popular visual authoring approaches in various domains.

Depending on ones point of view, one might opt to view a CI/CD pipeline configuration as a description of a flow graph. The property of being a graph offers the possibility to exploit well-known properties of graphs, such as reachability and establishing an ordering for nodes. A graph also lends itself very well as an underlying data structure for a visual representation. Nodes can be freely arranged on a canvas and connected with edges to symbolize the control flow.

Describing complex pipelines, workflows, processes, or similar structures is not a new problem, and has been subject to simplification via visual authoring in a number of domains.

The following sections give an overview over various visual authoring tools that employ a graphical DSL to guide the user to creating complex structures visually by composing smaller objects into large graphs. The fields investigated include 3D Modelling (Blender, Unreal Engine), Data Integration & Transformation (Azure Data Factory) and an already existing *visual editor* for CI/CD Pipelines (SemaphoreCI). GitLab has released a *visualization* of pipelines in early 2021 [31], which will also be discussed briefly.

Visual authoring has also been employed in other fields, e.g. teaching (MOOCat [3]) and training (VITA [20]), with the express purpose of making the design of complex processes more accessible to non-experts.

An in-depth comparison between the DSL proposed in Chapter 3 and SemaphoreCI can be found in Chapter 7.

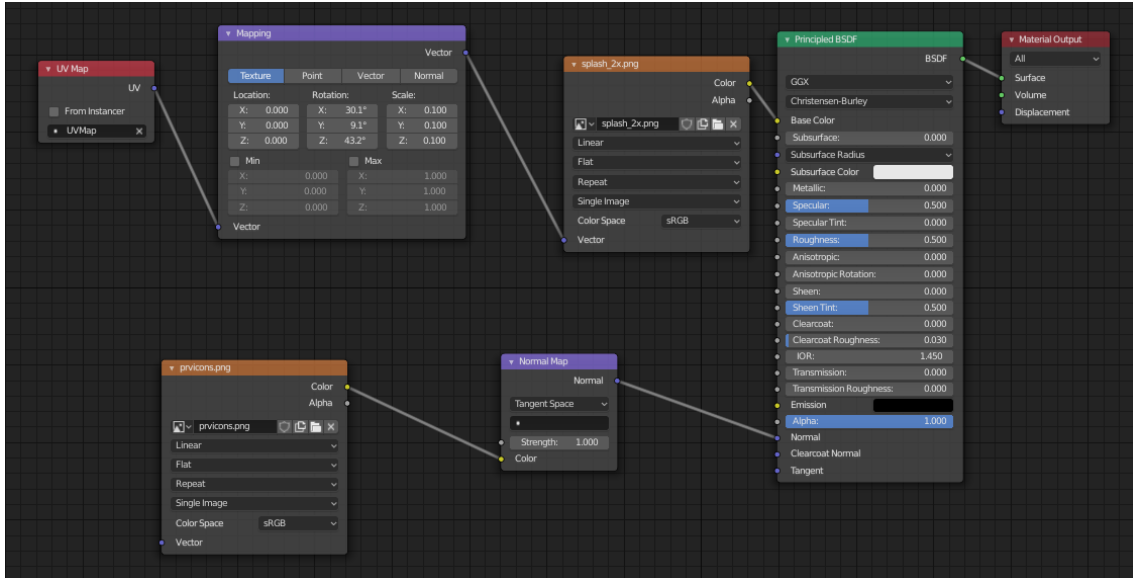


Figure 2.1: *Shader Nodes* in Blender [5].

2.1 3D Modelling

In the area of 3D modelling, where non-programmers often work together with developers to create compelling 3D scenes, either for movies, computer games, scientific visualizations, education or other purposes, visual authoring has long been used to make complex features more accessible.

2.1.1 Blender - Shader Nodes

Blender employs visual authoring to allow users to specify complex shaders via its so-called *Shader Nodes*. An example of such a shader node configuration can be found in Figure 2.1.

Of note in this case is that Blender uses a single target, labelled *Material Output* and located to the right of the image. Eventually, all nodes that contribute to the material must be connected via some path to this node. Orphaned nodes are possible, but do not contribute to the material and are functionally non-existent.

2.1.2 Unreal Engine - Blueprint Editor

Unreal Engine (UE) offers visual authoring for their so called *blueprints*. Blueprints are an alternative to writing code in C++ in order to describe the behavior of an application and offer a code-free way to develop complex applications within the engines SDK [15].

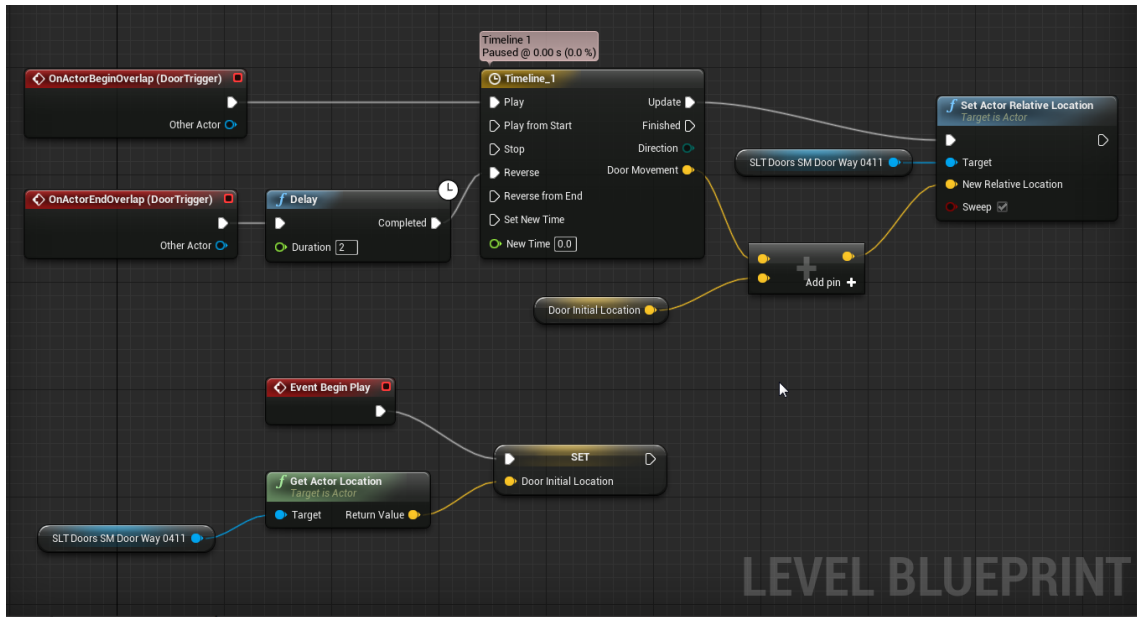


Figure 2.2: Unreal Engine Level Blueprint [16].

Since blueprints are intended to be code-free, the engine offers a powerful visual editor to configure those blueprints. An example of such a blueprint configuration can be found in Figure 2.2.

Of note is here that UE uses nodes for any input. Everything a blueprint can interact with or manipulate is loaded via a node, which allows for additional code-completion. Manipulating an actor is done by creating an actor node and then choosing the actor to be interacted with, and connecting that to further nodes who have such an actor as input and manipulate it.

In the lower left part of the aforementioned figure, the light blueish node is used to reference the actor named *SLT doors SM Door Way 0411*. This is used as an input for an "Get Actor Location" node. The *return value* of that node is used as input in the next node, a "Set" node setting a variable value. This happens when the "Begin Play" event is triggered, as modeled via the red node. The upper part of the blueprint is concerned with triggering the animation for the opening of the door when an actors bounding box overlaps with the door (the actors comes close to the door), and reversing it (closing the door) two seconds after there is no longer overlap (the actor is moving away from the door).

This gives the Unreal Engine a vast amount of power to perform sanity checks, e.g. to warn the user if they are about to delete an actor that is still referenced by a blueprint.

This strict model of requiring a complete description of the whole environment and all actors is particularly useful for the generalization of the DSL towards a platform-agnostic approach, as laid out in Section 6.2.

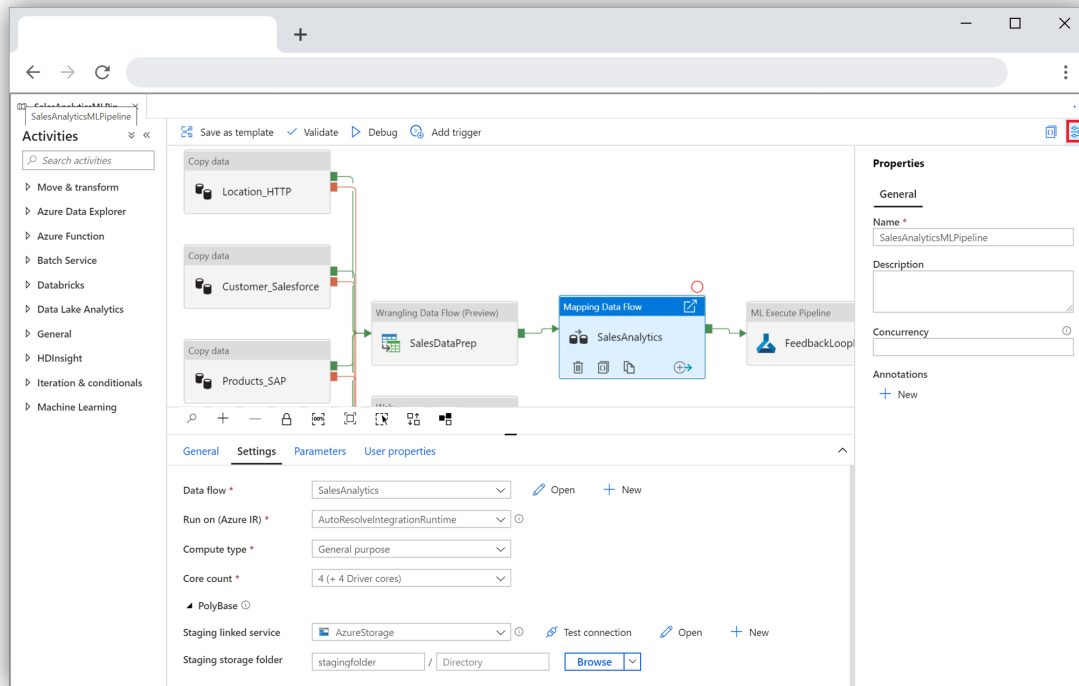


Figure 2.3: Data Transformation Pipeline in Azure Data Factory [35].

2.2 Other Fields

Visual authoring and graphical DSLs have not only be used to simplify programming as seen in the previous sections, but in other fields to make designing complex process more accessible. The authors of VITA note that "[g]raphical software has the advantage of being instinctively used by the users with few knowledge on computer science." [20]. In the following section, Azure Data Factory is looked at as an example.

2.2.1 Data Transformation: Azure Data Factory

Azure Data Factory (ADF) is "Azure's cloud ETL service for scale-out serverless data integration and data transformation.", and self-describes as offering "a code-free UI for intuitive authoring" [34].

An example for an ADF configuration can be found in Figure Figure 2.3.

Of note here is that the configuration is not done directly on the canvas, but on separate configuration panes outside of the actual canvas. The visualization includes a lot less information when compared to the Shader Node Editor offered by Blender or the Blueprint Editor of Unreal Engine. While the main workflow in terms of order of execution is visually accessible, the details are completely hidden and only available as settings.

This approach offers a lot of visual clarity. Both Blender and Unreal Engine come from a strong background in 3D modelling and gaming, and their UI is reminiscent of that. Azure Data Factory offers a visually less pronounced UI, with desaturated colors and a flatter look reminiscent of Material Design.

ADFs visual clarity and desaturated colors heavily influenced the look and feel chosen for Rig.

2.3 CI/CD Solutions

Visualizations and approaches to visual authoring are beginning to be used for CI/CD pipelines in commercial projects. Two such solutions are presented here: SemaphoreCI, a standalone CI/CD platform that currently only integrates with GitHub as repository host [49] and GitLabs Pipeline Editor, which is an ongoing effort to make CI/CD pipeline configurations more accessible [32]. Both of these solutions currently largely support *visualization*, with at most rudimentary support for visual authoring.

2.3.1 SemaphoreCI

SemaphoreCI is an existing CI/CD solution that offers a visual editor for their workflows. Figure 2.4 shows a CI/CD pipeline configuration authored in their UI. The platform offers ready-to-use templates for various scenarios; the shown configuration is based on their mobile app template.

SemaphoreCI uses the concept of so called *blocks*. Block can depend on other blocks. This can be configured via their UI under the heading "*Dependencies*". Blocks are then sorted and visualized on the canvas. The user can not arrange the blocks freely, the arrangement is done by the platform and can not be changed.

Blocks consist of *jobs*. All jobs in the same block are executed in parallel. By default, subsequent blocks are not executed when a block they depend on fails.

The configuration is done via the UI shown to the right. Interaction with the canvas is not possible. When the user is done, the configuration is converted to YAML and committed to the repository as `.semaphore/semaphore.yml`.

Much of the configuration is still text based. For example, artifact passing has to be configured as part of a jobs script block, using the platforms custom CLI tools. [40]. Linting is not available, and errors are only discovered when the file has already been committed to the repository and an erroneous run of the pipeline can be observed.

Parameterization of jobs is limited to using a value matrix. A disadvantage of this concept is combinatorial explosion. A job is executed for all possible combinations of variables that

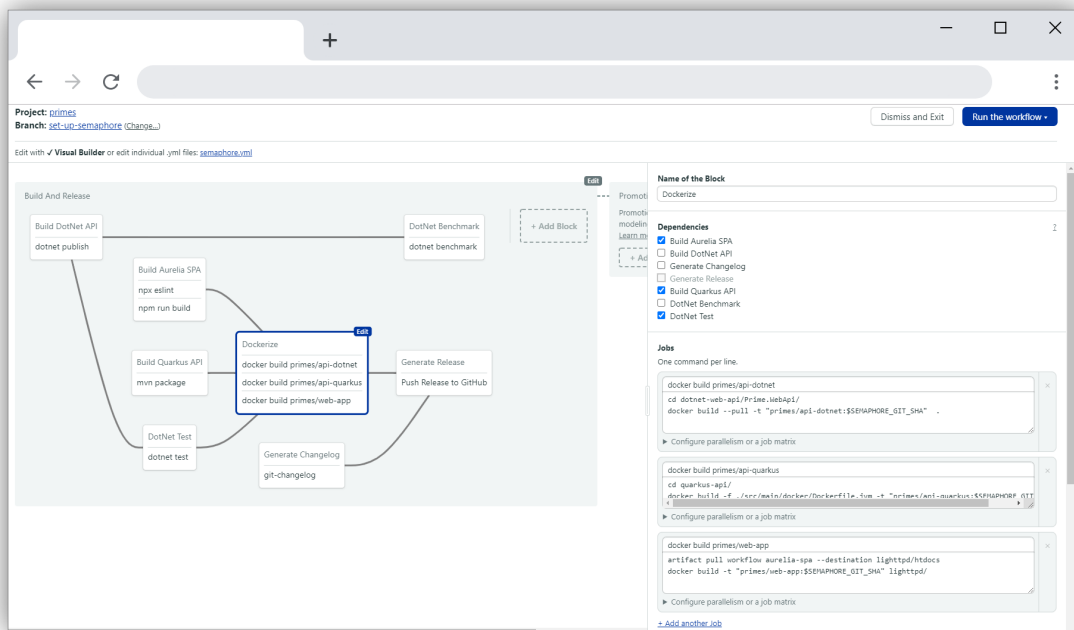


Figure 2.4: SemaphoreCI workflow for a mobile app; a template provided by the platform.

have been configured via the matrix. This not only leads to a lot of visual clutter, but also has the disadvantage that it is not possible to allow only a subset of all combinations.

Overall, the visual editor merely visualizes what is configured via the UI, but the visualization itself is non-interactive. Configuration is done via the UI to the right. There are no semantic or even syntactic checks for the input fields.

2.3.2 GitLab: Pipeline Editor

On Jan 22, 2021, GitLab released update 13.8 [31]. The update includes a so called "Pipeline Editor", featuring linting, configuration validation and a pipeline visualizer, as shown in Figure 2.5.

The novelty introduced in this update is that the pipeline is already visualized while editing the configuration through the platform's web interface. Prior visualizations that the platform used were only available for pipelines that were already executed. GitLab uses the concept as "stages", where jobs in earlier stages are executed earlier than jobs in later stages, but all jobs in the same stage are executed in parallel. The visualization used by the platform for these stages can be seen in Figure A.3. Unfortunately, this visualization does not properly reflect dependencies between jobs, and does not allow out-of-sequence execution (jobs are executed per-stage, not when the job they depend on is finished). To this end, GitLab introduced a new keyword, **needs**, with update 12.2 [38] [27]. This keyword allows jobs to execute once the prerequisites are met - thus jobs might not run

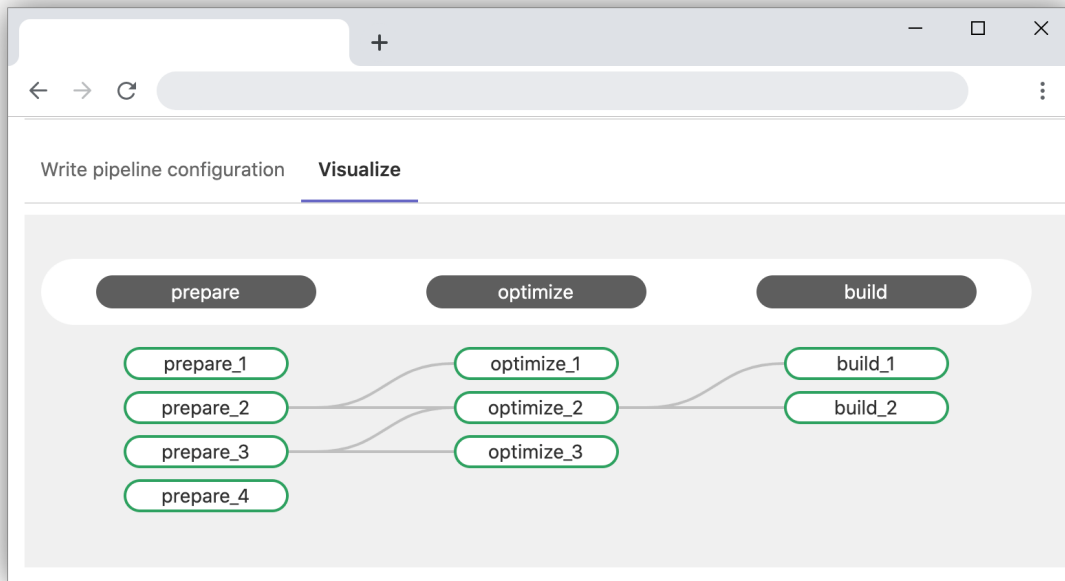


Figure 2.5: Pipeline visualization by GitLab, as shown in [31]

strictly in the order of their stages. A visualization of the dependency graph as given by the platform - again, only for pipelines that are already executing - in Figure A.2.

Both the newly added visualization and the prior visualization only show jobs and the dependencies between them, similar to the visualization of SemaphoreCI. The jobs and edges are arranged automatically, without the possibility of the user to interact with it. Configuration is still being done textually, via YAML. The resulting YAML is statically validated by a linter and a basic configuration check, but semantic errors can – in general – not be discovered by these tools.

An effort to add visual authoring capabilities to this editor is ongoing [32]. A preliminary mockup of these plans is shown in Figure 2.6. These plans do not (yet) seem to include bringing more visual authoring to the canvas itself, but instead focuses on providing these capabilities by providing UI elements like checkboxes, dropdowns and textfields in a side panel.

2.4 A Model-driven Approach

In [46], the authors propose a model-driven approach to continuous practices. A deployment model for a modern, cloud based web application is given in Figure 2.7.

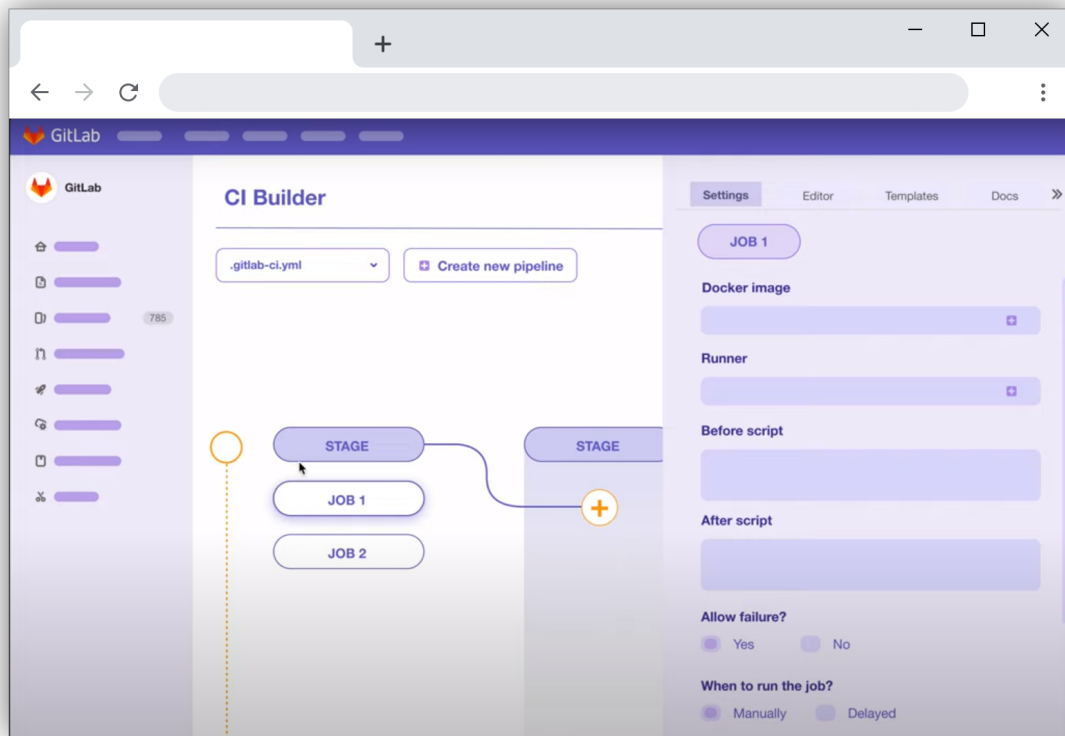


Figure 2.6: Mockup of a Visual Pipeline Editor, as shown in [45]

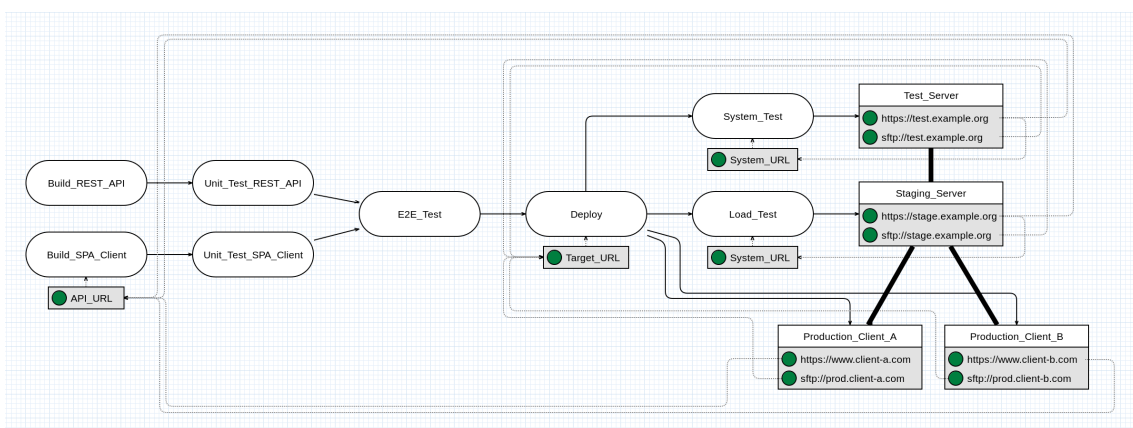


Figure 2.7: Deployment model for a modern web application, as shown in [46] (Fig. 2)

Of note in this approach is the possibility to assign jobs to multiple *build targets*, as well as the opportunity to parameterize jobs based on those targets, a feature which is largely unavailable or very limited in both Azure DevOps and SemaphoreCI.

The above model, combined with ADFs overall look and Blenders clarity and in-canvas editing capabilities have greatly contributed to the development of the DSL.

A key component of this approach is the concept of targets and their ability to parameterize preceding jobs. This allows adding new build targets without copying all the jobs needed to build a new target, especially when only minimal parameters of the build change, but the build process itself stays similar. Examples include targeting different hardware platforms or different deployments.

2.5 Important Takeaways

Parameterization of jobs can greatly reduce the complexity of the pipeline. By allowing the same job to be executed with different arguments, its is unnecessary to specify the same job multiple times.

The concept of targets as presented in [46] shall serve as basis for a fully fledged DSL. Existing CI/CD solutions largely lack this kind of parameterization, although efforts to reduce the complexity of pipelines have been made, e.g. by GitLab, which offers support for job templates that another job can inherit from.

Allowing the author to edit the workflow directly on the canvas – having a fully interactive visualization – greatly simplifies authoring of the pipeline.

Thus, making as many elements as possible editable on-canvas is preferable to hiding them in a UI that is detached from the visualization. Having model elements available for drag & drop onto the canvas will make features easily discoverable for users.

Chapter 3

A Graphical DSL for Visual Authoring

After introducing the problem space and taking a look at other related works, this chapter focuses on presenting a DSL that shall serve as the basis for a visual authoring tool that is capable of generating correct configurations for the target platform, GitLab.

3.1 CINCO - A Meta-modelling Framework

In order to create a DSL and visual authoring tool for CI/CD pipelines, the *CINCO SCCE Meta Tooling Framework* has been chosen as framework to develop the product – dubbed Rig – with. CINCO is marketed as "a generator-driven development environment for domain-specific graphical modeling tools [,] based on the Eclipse Modeling Framework and Graphiti Graphical Tooling Infrastructure" [9].

The elements of the DSL – the nodes and edges, and the relations between them, e.g. which edge may connect which node to which other node – are described in a language called the *Meta Graph Language* (MGL) [10]. Since CI/CD workflows are directed, acyclic graphs (DAG), they can be described well with this language.

The visual properties, e.g. fore- and background colour, border colour and size of the model elements (nodes and edges) are described in a language called the *Meta Style Language* (MSL) [11].

At least one MGL and an MSL file are tied together into a CINCO product using the *Cinco Product Definition* (CPD). The CPD serves as entry point for the generation of the so called *CINCO product* [8]. Product generation then generates all files needed to build an *Eclipse Rich Client Platform (RCP)* feature and application. While Eclipse is widely known as an Integrated Development Environment (IDE), the Eclipse platform is modularized to allow

developers to build arbitrary applications on top of it. The Eclipse RCP is "[t]he minimal set of plug-ins needed to build a rich client application" [14]. CINCO generates such a client application, called a *CINCO product*.

3.2 Preconsiderations

The initial goal was to describe CI/CD workflows in the context of GitLab. Thus, the DSL is tailored specifically to the features offered by GitLab. A more general approach for describing CI/CD pipelines is briefly discussed in Section 6.2.

CI/CD pipelines generally work by executing so called jobs in order. On most platforms, including GitHub and GitLab, jobs can specify the Docker image they are run on (choosing a default container absent explicit configuration). The provided script thus has to be compatible with the OS and shell used in that image (e.g. ‘sh’ for most Linux-based images). Since multiple jobs might depend on the same environment (for example, in a project using .NET Framework, a job might produce an artifact – a DLL – using `dotnet publish`, and a subsequent job might want to test the same artifact using `dotnet test`, and will have to do so using the same SDK version), making this property a shareable element that can be applied to multiple jobs at once is preferable. This reduces redundant work and reduces the potential for errors. For example, when updating the image to a new version, one only has to change the version number once, and all connected jobs will use the same version number, taking away the opportunity for mismatching versions being used.

The order of the jobs is given by evaluating their dependencies. Jobs can depend on previous jobs and may re-use artifacts a previous job has produced. In the aforementioned example of a project using .NET Framework, a job might want to run `dotnet test` on the DLL produced by a previous job that created that kind of artifact via `dotnet release` or `dotnet publish`. This artifact has to be passed from the job producing it to the job consuming or processing it.

Artifacts can also be used to release a product or to directly deploy it. As such, the DSL must support artifacts for passing between jobs, for releasing a certain software version, and for deployment.

Jobs might not only want to specify the Docker image they are to be executed on. Integration tests may need additional resources, e.g. a database server. As such, GitLab supports the configuration of additional services. Services may be shared by multiple jobs, thus making them a shareable element that can be applied to multiple jobs at once the preferred option. The same considerations wrt. updating images also applies to services.

Another large consideration is the conditional execution of pipelines and jobs. By default, GitLab executes jobs on every commit and every tag. Users might want to constrain this further, e.g. by allowing some jobs only to be run on certain branches or on certain tags, but they also might want to use completely different conditions for execution. GitLab supports a multitude of so called *events* that can trigger a job. Those events include API calls, chat events, merge requests and more [27]. GitLab offers three mechanisms for constraining the execution of jobs - the **except** property which constrains job execution to all conditions *except* the ones listed in the configuration, the **only** property, which works analogous, but *only* allows job execution under the circumstances listed in the configuration, and a more advanced **rules** property, which uses its own binary relation syntax to evaluate conditions. While **except** and **only** can be combined, they are incompatible with **rules**. Since GitLab was chosen as target platform, the DSL should be able to support conditions in these three forms.

Several elements (aside from jobs, targets and the aforementioned elements) have been identified as being suitable for modeling as a *node* in the DSL. The elements to be used in the DSL are: Artifact(s), Cache, Release, Image, Service, Except & Only, Rules, Environment and Retry. All of these elements are called *properties* in the following, since they are properties a job *can have*. This relationship will be modeled as an edge from the property to the job in the DSL. The user can simply drag an edge from a property to a job and connect them to make the property apply to that job.

3.3 Approaching Polymorphic Values

GitLab often exploits the inherent lack of type-safety in YAML (which it inherited from JSON and subsequently JavaScript) to provide a user with multiple configuration options.

For example, the **only** property can take either a scalar value, an array of scalar values, or a single hash (a dictionary) as value. Great care has to be taken to ensure that reasonable configurations can be generated with the DSL.

While treating scalar values as a single-element array (or vice versa) is trivial, hashes have to be given individual treatment to ensure they have reasonable semantics. For example, the **image** property can only appear once per job, while the **only** property may appear multiple times on the same element. If a job has multiple instances of the **only** property, these properties can easily be merged into one for code generation (cf. Figure 4.1).

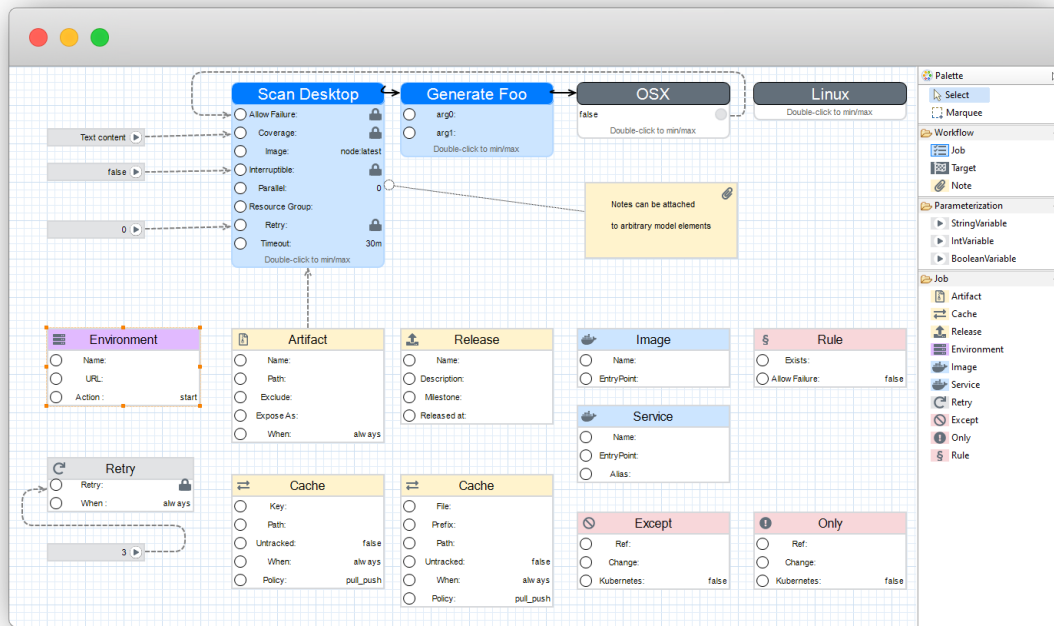


Figure 3.1: The complete palette of the proposed DSL

3.4 The Elements of the DSL

The DSL consists of four main elements - jobs, targets, variables, and properties (c.f. Figure 3.1).

While properties with scalar values can be configured directly on the job, hashes (dictionaries) are standalone elements which get connected via an assignment edge to either a job or a target. Jobs can be connected to other jobs to establish their order of execution.

To parameterize jobs, targets offer two mechanisms - parameters and assigned properties. Assigned properties are propagated to each job reaching the target, while the parameters (which are scalar values) assigned to a target can be connected to individual scalar properties, but only for those jobs that actually do reach the target. It is an illegal operation to connect parameters to a property that isn't either part of a job that reaches the target or assigned to a job that reaches the target.

If multiple variables are connected to the same property, the property is considered to be an array of scalar values instead of a single scalar value. Properties that can be either a scalar value or an array of scalar values can be specified zero, one or multiple times on the same element.

Properties, variables, and parameters are typed to ensure that only valid values are assigned to each property. For example, the `Retry` property is implemented as an `IntegerProperty`, which only allows assignments of edges of the type `IntegerAssignment`. Those assignments

can originate from the elements `IntVariable` (a variable that can freely be placed on the canvas) and `IntParameter`, an element that can be added to a target to allow job parameterization depending on the target. Three types of primitives are supported – `int`, `string` and `boolean`. Where more validation is needed, semantic checks were implemented via the corresponding CINCO meta plugin or as a separate XText grammar.

In the following paragraphs, some of the notable elements of the grammar (and the reasoning behind their existence) are explained.

Jobs Of the properties that can be added to a job, several are not just scalar values, but can also be arrays of scalar values. These properties can be specified multiple times and are `Tag`, `Resource Group`, `Except`, and `Only`. A special role not found on other elements is filled by `Script Arguments`. This property can also be specified multiple times, but each script argument has to have a unique name. Script arguments are central to job parameterization, as they will often be filled via target parameters.

`Except` and `Only` are the most complex of these properties. While they were designed to be very easy and intuitive to use, the complexity lies in the fact that both of these properties can also be a single hash (dictionary) value. If an `Except` or `Only` property is assigned to the job, then the scalar values added to the job are considered to be part of the `Ref` value of the hash, allowing for free mixing of matching of scalar conditions and the proper elements of the DSL.

Targets Targets are used to parameterize jobs. For each target a job leads to, the proper configuration for the CI/CD pipeline is to be emitted, taking into account the parameters of the target and where they connect to.

They accept some of the properties that can usually only be assigned to jobs. This includes all of the conditional properties (`except`, `only` and `rules`). The semantic of this is that these properties are to be applied to all jobs leading to the target. This way, conditions that should be common for all jobs leading to that target can easily be specified. It is also an intuitive semantic to allow *targets* to be conditionally executed.

The same consideration applies to both `images` and `services` - the two other properties that can be applied to targets. By allowing them to be specified for targets, one can easily set the default image for all jobs leading to a target. If an image is also configured for a job leading to a target that already specifies an image, the image specified on the job takes precedence.

This feature is helpful in many different situations - from specifying different OS images or platforms to built on in order to target multiple systems, to allowing multi-project repositories. A *Backend* target might configure the needed container image needed to build

the backend - e.g. Quarkus or .Net Framework, while a *Frontend* target might configure a different environment, e.g. Node.js.

Notes YAML files allow the user to add comments to their configuration - a feature that is often used to document or explain the build process and to leave hints for future developers (or oneself in the future). And while a proper graphical DSL can already serve as documentation tool for the build process – especially when it not only visualizes the process by displaying jobs and their dependencies, but includes all relevant elements directly on the canvas – documentation of the build process is still far from complete with only such a graphical representation. None of the tools studied in Chapter 2 include the possibility to annotate the configuration or add comments.

UML notably includes a comment element that "add[s] no semantics but may represent information useful to the reader of the model" [37] (p. 22). Such an element can be very helpful in explaining design decisions and the reasoning behind the structure of a CI/CD workflow.

While the primary goal of the DSL was to make writing configurations easier and the process more accessible, a graphical representation of the pipeline has also been found to be a very good communication tool to explain the build process. In order to promote this self-documenting aspect of the DSL, a **Note** element has also been included in the DSL. The note allows the user to specify free-form text which is displayed within the rectangular area of the note. Notes can be resized, freely placed, and can optionally be connected to arbitrary elements of the DSL to highlight for which element they provide documentation or simply commentary.

Notes can be freely colored by the user to highlight different areas. The default color – a desaturated yellow – is reminiscent of Post-it™ Notes, while visually not overloading the representation. A paperclip as icon reinforces the notion of being an addition, a tacked on comment.

Other properties The other properties largely follow the configuration options given by GitLab. Complex properties that are configured as hashes are represented by the accompanying model element. These model elements can either be connected to jobs or targets. All of the *conditions* (**rules**, **only**, **except**) as well as environmental properties such as **image** and **service** can be assigned to targets, with the semantic meaning that they will affect every job leading to that particular target. All other properties can only be assigned to jobs. Depending on the property, a property assigned to a job either takes precedence over a property assigned to a target or can be merged with the target property. These mechanics are further discussed in Chapter 4.
















Icon	Name ¹	Model Element	Accent Color
	tasks	Job	Blue
	flag-checkered	Target	Gray
	play-circle	Variable	Light Gray
	lock	Property Value ²	Dark Gray
	paperclip	Note	Light Yellow
	file-archive	Artifact	Light Yellow
	exchange-alt	Cache	Light Yellow
	upload	Release	Light Yellow
	server	Environment	Light Purple
	docker	Image	Light Blue
	docker	Service	Light Blue
	redo	Retry	Light Gray
	ban	Except	Light Red
	exclamation-circle	Only	Light Red
	Section Sign ³	Rules	Light Red

Table 3.1: Icons and color palette as used by Rig

3.5 Color Scheme & Iconography

Colors and icons are important guides for users and can greatly enhance user experience by guiding the user intuitively towards the proper elements. The colors and icons in to be used with the DSL in Rig have carefully been chosen to promote ease of use. The following sections show the design decisions and considerations used to determine the color scheme and proper icons used for each element.

3.5.1 Color Scheme

The elements of the DSL are grouped together by color. Jobs appear in a bright blue, since they are arguably the most important elements of the pipeline description.

Parameterization elements use a gray color. Targets are dark gray, while variables use a very light gray. This gives more visual prominence to targets than variables, since targets are the primary parameterization elements, and should thus stand out more.

¹As used by Font Awesome

²*Locked* for direct value inputs due to being parameterized by a variable or parameter

³Unicode code point U+00A7, rendered in "Segoe UI" font

Properties use desaturated colors to group them together by function. Properties that control the execution via conditions are color red - **Except**, **Only** and **Rules**. These properties will restrict the execution of jobs until certain conditions are met, and are thus prominently colored. Properties related to the environment use a desaturated blue, these are **Image** and **Service**. The former describes the Docker image used for the execution of the job, the latter describes further images for Docker containers that need to be present during the execution of the job to provide additional services, e.g. a MySQL database. The **Environment** property uses a desaturated magenta since it both describes the environment (for deployment) and conditions under which certain further actions are triggered. A desaturated yellow is used for properties that describe meta-information about the build, like caching, where the build process produces an artifact, and the corresponding release. It is also the default color for notes.

The color scheme and the ordering of elements in the pallet makes it easy for users to find the right elements quickly and to see which elements are semantically related. Red is a color often associated with STOP-signs, danger or warnings, and was thus chosen as color to represent conditions that restrict or allow the execution of the pipeline, making these properties stand out at first glance. With jobs being the visually most pronounced elements by having a stronger, not desaturated header color, the eye of the user is first guided to jobs. Being red, the next element to catch the eye are the conditions. Thus, the questions of *what* and *when* are the ones first drawn attention to. The question of *what for* is answered next by drawing attention to targets, which also use a richer, not destaturated header color. All other supplemental information is still easily visually accessible, but uses destaturated color to not overwhelm the user visually and to take a backseat. Thus, the primary structure of the pipeline can easily be gleaned by looking at the diagram, and more details can be discovered by taking a closer look.

The primary colors chosen for this DSL were greatly inspired by the colors of Twitter Bootstraps (TWBS) Alert components [6]. The color for targets and jobs was inspired by Azure DevOps Data Factory (cf. Figure 2.3).

Overall, the design decision to mainly use desaturated color was made in order to avoid visually overwhelming the viewer. During testing, more pronounced colors were used, but ultimately discarded due to their cognitive load and unsuitability for this application.

3.5.2 Iconography

The icons were carefully chosen from the collection of Font Awesome⁴ due to being available under free, Open Source licenses⁵, and the need to have a large catalogue with icons

⁴<https://fontawesome.com/>

⁵<https://fontawesome.com/license/free>

representing a variety of concepts, together with the authors familiarity with the catalogue. In the following, the rationale behind each icon choice is laid out in detail.

Job The icon chosen for jobs is a partially completed task list, representing the step-by-step shell execution of the jobs script commands.

Target The icon for targets was taken from sports, being checkered flag used in e.g. Formula 1 as flag representing the races finish line.

Notes Notes, which can freely be added to the diagram and only offer supplemental information, are represented by a paper-clip, representing notes physically being added to papers, books or folders on an actual desk.

Artifacts Artifacts show a generic archive symbol, representing the fact that artifacts are usually packed as a single archive in various formats (mainly ZIP or TAR) which facilitate easy transfer of multiple resources from one device to another.

Cache The cache is represented by a double-sided arrow, representing both the push and pull direction in which a cache can operate.

Release The icon chosen to represent the release property is the generic upload icon, due to the fact that the release of a software usually involves uploading the finished build artifact to a (public or private) location for further consumption by consumers (either end users or internal users, e.g. quality assurance teams).

Environment A generic server symbol alluding to a rack in a data center has been used to represent the deployment environment. The rationale was evoking a feeling of deploying the application to actual hardware somewhere in the world.

Image & Service Both images and services, as used by GitLab, are given as Docker images. Hence the Docker logo has been chosen to represent these properties.

Retry The retry property is used to execute jobs multiple times when they fail, until the jobs succeeds or reaches the maximum allowable number of retries. The act of doing the job again has been represented by the "redo" icon, which is commonly used in applications to represent the act of redoing something (usually previously undone).

Except The except icon uses a generic "ban" or "forbidden" symbol, representing the fact that it forbids the execution of the pipeline under the given conditions.

Only The icon used for only is an exclamation mark or warning sign, highlighting the fact that pipeline execution may *only* occur under those notable circumstances given by this property.

Rules The paragraph sign (also known as section sign) is commonly associated with paragraphs in laws, rules and regulations, and thus has been chosen to represent the rule property.

Variables A triangle pointing to the right, toward the socket from which connections to properties are made, was used to illustrate the DSL element for variables. It signifies the data flow from the variable towards the property it is connected to.

Furthermore a *padlock* icon is used to replace the value display of properties when a parameter or variable connects to the property. By replacing the value with this icon, it is signified that the value can not be edited in-place, but stems from another source and has to be changed there in order to affect the property. The local value is no longer used, and the value is "locked" for editing.

By choosing concepts that are related to (or can metaphorically be related to) each element, the user is provided with additional means to identify the purpose of each element of the DSL.

Chapter 4

Deriving Configurations from the Graph Model

The code generator shipped with Rig is capable of transforming models authored using the DSL into the proper YAML configuration for the target platform, GitLab.

The first approach investigated for code generation was mapping the graph model to a domain model, and then converting the domain model to YAML, either via XTend templates or preferably a YAML mapper.

Since CINCO produces an Eclipse RCP application, the code generator for Rig is implemented in Java. Jackson [51] has been chosen as JSON library due to its YAML support and support of Java 8 language features, something contenders like GSON lack [50]. Using Jackson means the focus could be on transforming the model from the DSL to Jackson’s abstract representation of JSON, without having to deal with the syntactic intricacies of JSON. Jackson correctly escapes all input values and outputs them in proper JSON format, a non-trivial task if done by hand. By leveraging the `jackson-dataformat-yaml`¹ package, the same guarantees can be obtained for YAML output.

However, the transformation of the graph model to a domain model would have resulted in substantial implementation work, since a lot of the bookkeeping (mostly edge connections) has to be duplicated in the domain model. This approach, while straight-forward conceptually, has since been disfavored due to the huge amount of boilerplate and thus implementation effort required to both initially write – and in the future maintain and evolve – the code generator.

A considerable breakthrough in the area of code generation was achieved by studying the model-to-model transformation approach using an variation of SOS as laid out in [33]. In

¹<https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-yaml>

their paper, Kopetzi et. al. describe a variation of Plotkins SOS to facilitate pattern-based transformations of typed graphs. The authors introduce graph patterns as well as node and edge types – features which are easily applicable to CINCO models.

Studying this approach, it became clear that a direct graph-to-YAML transformation was feasible. At first, transformation rules within the graph model were studied in an informal fashion. A sketch of the rules can be found in Figure 4.1. In this sketch, auxiliaries and conditions have been omitted and only source language patterns and target language patterns have been included. The names of the rules are not chosen rigorously, but represent a best-effort approach to naming.

A more detailed explanation of the rules and their intended semantic, as well as their pre- and post-conditions follows below.

Property merging The elements `only` and `except` can be merged by simply merging their keys into arrays, since GitLab matches an `only` property when *all* keys match and an `except` property when *any* key matches [27], giving exactly the desired semantic. These elements can only be merged with each other, respectively, not with other elements. Other elements for which merging strategies can be found are artifacts and cache. The `image` element can notably never be merged, as a Docker image is uniquely identified. The rule is shown in Figure 4.1 for the `only` element.

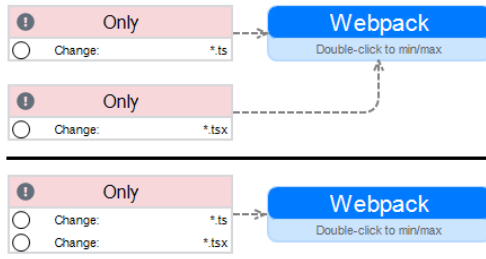
Property unzipping A property which is connected to multiple jobs must be duplicated and connected to each of these job, individually.

Target splitting This rule proved instrumental in implementing a code generator. It can be used to fulfill the pre-requisites of the property propagation, parameter propagation and target elimination rules, which require sub-graphs with only one target each. While this rule was changed significantly for the actual code generator, the concept of only considering one target and the relevant sub-graph is what enabled implementation of a code generator with a reasonable amount of work.

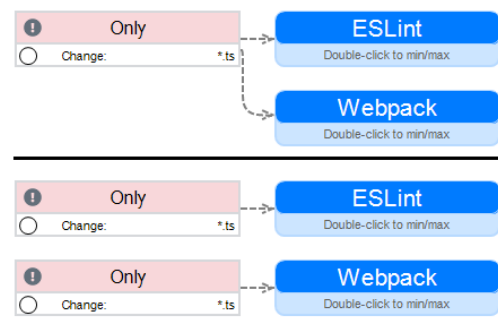
Property propagation In subgraphs with only one target, properties assigned to the target can simply be added to each and every job that eventually reaches the target, and then removed from the target.

Parameter propagation This rule works analogous to the property propagation rule, but for target parameters.

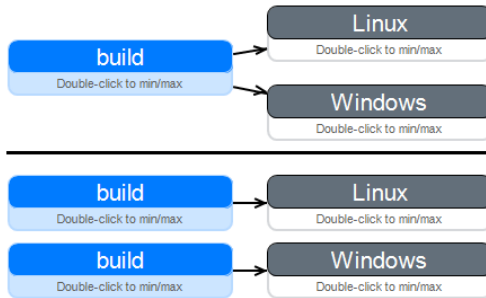
Target elimination In subgraphs with only one target (which can be produced by applying the target splitting rule) and without any parameters left that need to be propagated (and no properties assigned to the target), the target can be eliminated altogether. In order to still be able to discriminate jobs, the name of the target is



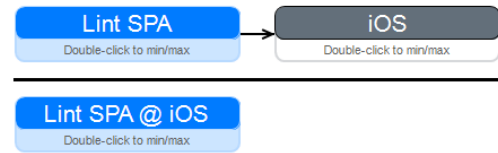
Property merging (only)



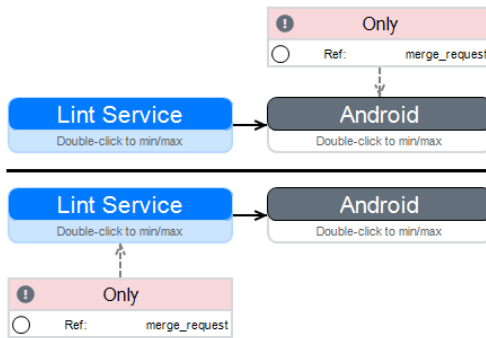
Property unzipping



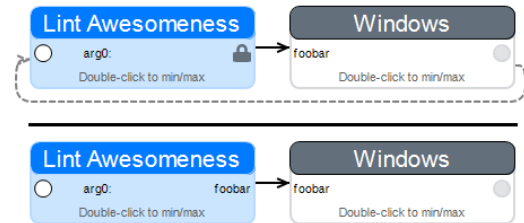
Target splitting



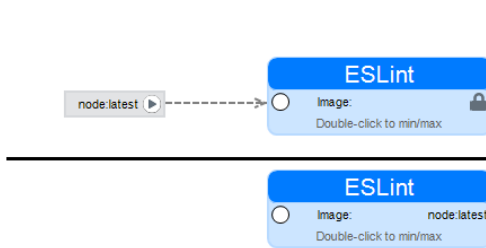
Target elimination



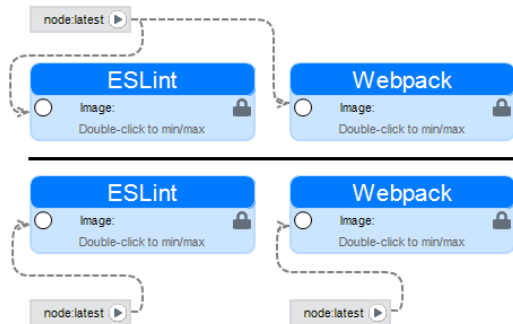
Property propagation



Parameter propagation



Variable inlining



Variable unzipping

Figure 4.1: Informal sketch of the studied transformation rules

added as suffix to each job, with the @-sign serving as delimiter. Parameters and properties can be propagated by the preceding two rules.

Variable inlining When processing a job or property, the values found in any assigned variable can simply be incorporated into the model element by copying their literal value and eliminating the edge and variable. Prerequisite for this rule is that the variable only has one outgoing edge. this can be ensured by the following rule.

Variable unzipping A variable that is connected to multiple properties can be duplicated and the edge transferred to the duplicate. Repeated application of this rule leads to variables with only one edge, which can then be inlined and eliminated with the preceding rule.

These rules already compose very well and allow reasonably complex workflow to be transformed. An example for the composition of the property unzipping and merging rules can be found in Figure 4.2.

Studying these transformations, it became clear that a direct transformation of the graph model to YAML was in reach and could be achieved without the burden of re-implementing a significant amount of the bookkeeping logic in the original model. The model-to-model transformation rules shown above can easily be re-written in Java to target the JSON metamodel that Jackson offers.

As mentioned, especially the target splitting rule proved instrumental to achieve acceptable implementation effort. The rule highlighted the fact that one can derive configurations from the model partially, by only taking one target and the sub-graph leading to that target into account. Thus, the targets can be added to a working set, and then all relevant jobs can be processed by walking the relevant sub-graph backwards, until no other preceding job can be found. Since only one target at a time is considered, only the parameterizations from one target (and from variables) have to be considered. For variables and properties, the transformation rules laid out above translate in a very straight-forward way to Java code transforming the given model elements to Jacksons JSON metamodel. The property merging rule translates to pushing elements to JSON arrays in a single property, the property unzipping rule does not need to be done explicitly, but is implicitly achieved by looking up properties connected to a job instead of navigating from properties to jobs. Variable inlining is trivially achieved by looking up the variables connected to a jobs properties while processing the job, and analogous for properties.

Thus, code generator works by walking the graph backwards, starting from targets. The model elements that can be applied to targets – the conditions (**except**, **only** and **rules**), as well as the environmental properties (**image** and **service**) – can easily be closed over for propagation through the whole sub-graph, achieving the property propagation and

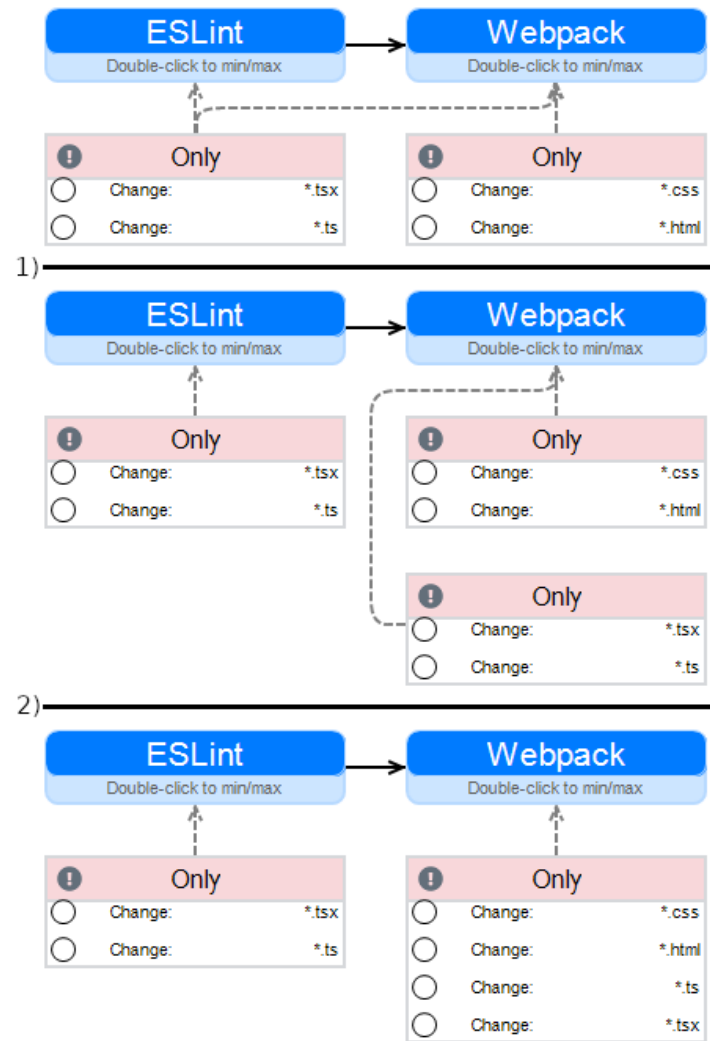


Figure 4.2: Composition of the property unzipping (1) and property merging (2) rules.

parameter propagation rules. The algorithm can then recursively walk through the jobs connected to the target, navigating by the edges connecting jobs. When processing a job, the algorithm emits a job consisting of the job name in the model, followed by an @ sign, and the name of the target. This way, each job is emitted once for each target. This solution is not minimal. A job that is not parameterized can be used for multiple targets. Minimizing the amount of jobs emitted by the generator is possible by inspecting the jobs for incoming parameter assignments, and only emitting a job per target if needed. This has not yet been implemented. Notably, the transformation rules as presented do also not lead to a minimal model. A more rigorous investigation into these kinds of transformation is expected to yield minimal models.

When processing a job, all incoming edges that belong to properties are evaluated and the proper configuration is emitted for each job. Properties themselves can have incoming edges, either from variables or target parameters, so these are taken into account as well.

Some properties can occur multiple times. As explained in Chapter 3, an `image` property applied to a job takes precedence over an `image` property applied to a target. The conditional properties have to be merged. This can be done by merging their keys, since GitLab already properly considers multiple values for `except` and `only` by merging them as conjunction or disjunction, respectively. Similarly, the `rules` property can also be merged into a single property and be emitted.

The code generator collects all stages encountered during walking the graph, and emits the stage configuration after processing all jobs. Since the stages are derived automatically from the graph model, the user does not need to concern themselves with configuring the proper stages.

By basing the code generator on a theoretical framework of transformation-based rules, it ended up being fairly concise - about 850 lines of code (LOC) distributed over two classes. This is smaller than the MGL itself, which comes in just short of 1000 LOC. A contributing factor to the size explosion of the MGL is the amount of repetition required for similar elements, especially wrt. annotations, which are not inherited by child elements, of which the MGL makes heavy use.

Chapter 5

Results

The DSL as presented here offers all features to create real-world CI/CD pipeline configurations. The code generator, albeit still crude, already works as expected and generates reasonable YAML files, although they are not yet minimal.

In the following section, a cloud-based web application with a Java-based backend using Quarkus and a frontend consisting of a single-page application (SPA) using Aurelia is used as a case study for the DSL.

5.1 Case Study: A Modern, Cloud-Based Web Application

Figure 5.1 shows an example of a CI/CD pipeline configuration for a cloud-based web application. In this example, Quarkus - a Java framework - has been chosen for the backend and Aurelia - a JS/TS framework - has been chosen to develop the SPA for the frontend. Both frontend and backend are packaged as Docker image for easy deployment. However, the workflow also produces both the artifact (a JAR) needed to run the backend directly as well as an archive (TAR) containing the built SPA. As such, it can also be deployed without Docker or delivered to customers with their own servers, and does not necessarily rely on Docker.

The shown workflow demonstrates most capabilities of the DSL. To keep the workflow small, caching is not demonstrated. Of the properties related to runner configuration – `image` and `service` – only `image` has been used in this workflow, and of the three conditions (`only`, `except` and `rules`), only one is used here. The `retry` property has not been incorporated.

The same *Image* element has been used to configure both the *Lint* and *Build SPA* job. The image chosen in this example is Node.js 12 running on Alpine Linux¹. By using the same

¹ https://hub.docker.com/_/node

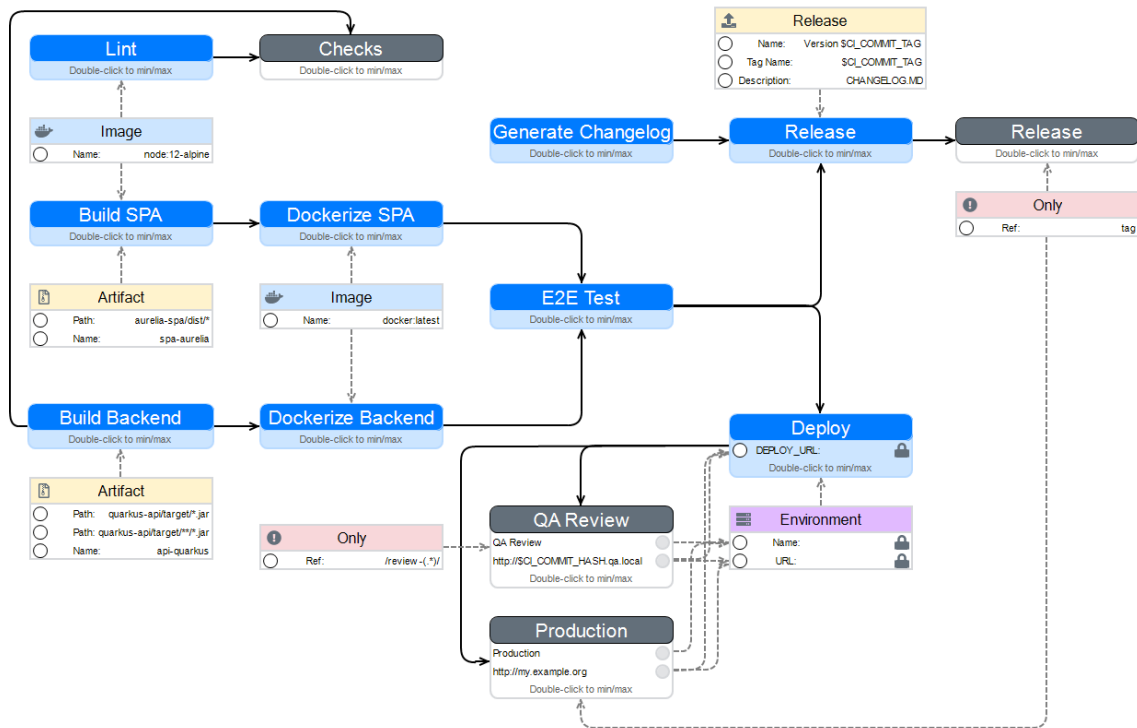


Figure 5.1: A cloud-based web application deployed via Docker, using Aurelia as SPA for the frontend that consumes a web API backend implemented via Quarkus.

element on both the linting and building jobs, switching to new node versions is easy, and the danger of missing updating a job is mitigated. Updates to the image always affect both jobs and thus the jobs are updated in lockstep. this also makes switching the platform to easy, if Alpine is at one point no longer desirable.

The same modeling technique, with similar benefits, is employed for the Docker image used by the two jobs that package frontend and backend into Docker images. While having two different versions of Node.js for the linting and build job might be a mere inconvenience, having the same Docker image (including version) for both images is of much greater importance, since incompatibilities between the Docker image versions could lead to the inconsistencies.

Since Quarkus is used for the backend, separate jobs for unit and integration tests of the backend are superfluous – Maven already runs these tests during the build job via the `maven-surefire` plugin and `maven-failsafe` plugin.

The workflow uses four targets – *Checks*, *Release* and the targets used for deployment, *QA Review* and *Production*.

The *Checks* target is run on every change and ensures that unit and integration tests of the backend do not fail. Since unit testing the UI was omitted from this workflow, only the

linter is run for the SPA. This ensures compliance with all tests in a fast-moving commit cycle and that any problems in committed code are caught early.

The *Release* job is used to capture the binary files associated with each versioned release and to generate a changelog. These binary files can then be distributed to third parties, e.g. customers who choose to host the application themselves. The binary files can also be used to re-deploy the application in exact the same version, should the need arise.

The *QA Review* target is only executed for refs that match the expression `/review-(.*)/`. This allows for the targeted use of QA (Quality Assurance) resources, which need not necessarily be involved in every single commit.

Finally, the *Production* target is used to deploy the application to the live server. Notably, the same *condition* is used for the *Release* and *Production* targets. This insures that there is always a proper release generated for every version that gets deployed, so no deployment goes undocumented. By using the same element of the DSL, the condition can be updated to match new requirements, but it keeps both the Release and the Deployment always in lockstep.

Adding new deployments to such a configuration is straight-forward, and one of the great strengths of the DSL. A new target can easily parameterize the *Environment* element and the *Deploy* job to deploy to a different environment.

5.2 Summary

The DSL is capable of expressing many concepts relating to CI/CD pipeline configuration, especially those relevant for CI/CD pipelines for the target platform, GitLab.

Targets allow the easy parameterization of jobs, elements of jobs or even entire sub-graphs. This significantly reduces the complexity of the configuration, albeit possibly at the cost of a lot of edges flowing from targets through the graph. Possibilities to reduce edge proliferation are discussed in Section 6.4.

The ability to re-use model elements on different jobs greatly reduces the complexity of the pipeline and reduces the danger of accidentally introducing inconsistencies. By using the same element on different jobs, updates to the element are automatically applied to both jobs, preventing them from getting out of sync. This can help keep OS versions, tool versions, conditions and more consistent between jobs when updating to newer versions.

Chapter 6

Future Work

The DSL – as presented so far – and the accompanying code generator already ensures that only *syntactically* valid *YAML* is generated.

However, the DSL still contains free-form text-inputs to provide *values* for various properties, e.g. regular expressions. Some of these values are already supported with XText grammars to help the developer write correct inputs e.g. the expression grammar used for `rules:if`.

Work on Rig to implement more of these low-level checks and grammars is still ongoing, while many of these checks can be eliminated (made obsolete) altogether by a better abstraction. Opportunities for making semantic checks for (pre-) conditions obsolete by interpreting the DSL appropriately are discussed in Section 6.1 . In Section 6.2, the possibility of the generalization of the approach – removing the hard dependency on GitLab – is explored and it is shown how this can lead to a much more abstract, powerful, and easier to reason about DSL. Linting regular expressions can be made obsolete by employing visual authoring for this problem as well, as discussed in Section 6.3. In Section 6.5 and onward, the focus is laid on improving the quality of Rig by adding quality of life features on top of a powerful DSL, creating a fully fledged IDE for visual workflow authoring in the process.

6.1 Ensuring Instantiation of Prerequisite Jobs

A job that depends on a prior job (via `dependencies: ["Other Job"]`) can only be instantiated when the job it depends on also has been instantiated. Listing 1 shows a syntactically valid pipeline. Figure 6.1 depicts how an equivalent configuration can be visually authored using the DSL.

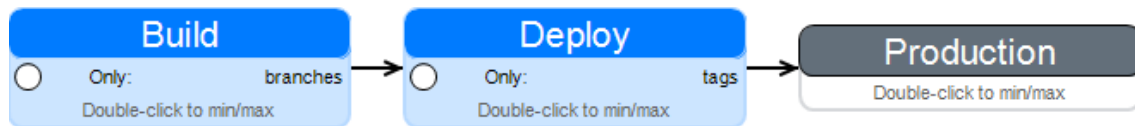


Figure 6.1: Invalid configuration; leads to failing pipeline when creating or pushing a tag.

```

1 Build:
2   script: echo "Building ..."
3   stage: build
4   only: [branches]
5
6 Deploy:
7   script: echo "Deploying ..."
8   stage: deploy
9   only: [tags]
10  needs: [Build]
  
```

Listing 1: Syntactically valid pipeline configuration that appears to work when pushing branches, but fails when pushing tags.

When GitLab attempts to instantiate a job of which the predecessor has not been instantiated, the pipeline fails. The shown configuration produces the error "'Deploy' job needs 'Build' job, but it was not added to the pipeline", tagged with the labels "yaml invalid" and "error", as seen in Figure 6.2.

In the example above, the aforementioned error will only be seen the first time a new tag is added to the git repository. As long as only new commits are pushed, the pipeline will run without error. The CI Lint tool will accept the configuration as valid. But as soon as a tag is created, the pipeline run will fail with the aforementioned error. This delays discovery of the error potentially days, weeks or even months after seemingly correct changes to the configuration have been committed to the repository.

This also means that pipeline configurations are not *composable*. If a job, which is correct when looked at in isolation, is added to another, correct configuration, and becomes a dependant of an existing job in that configuration, the resulting configuration may *seem* valid at first glance, but might not work correctly once the correct circumstances arise.

Copying a working job from one working configuration to another working one thus might lead to an incorrect configuration. The failure of the GitLab platform to adequately handle this scenario breaks composability of configurations in a very significant way.

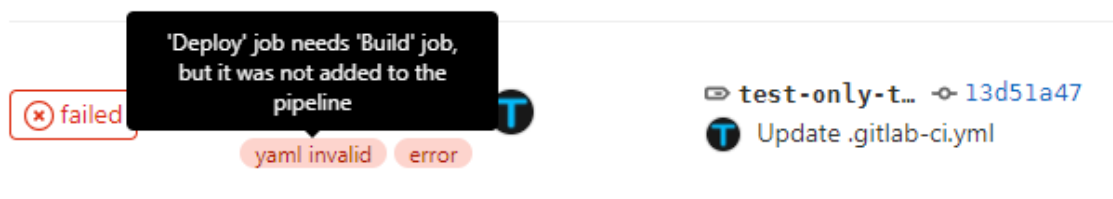


Figure 6.2: Invalid configuration; failing pipeline due to incorrect dependencies when creating or pushing a tag.

This error is particularly cumbersome due to particularities of the GitLab platform surrounding scheduled pipelines.

GitLab allows pipelines to be scheduled in intervals [28]. This is useful e.g. to make nightly builds. Jobs can either be included in or excluded from the pipeline via `only: schedules` or `except: schedules`. The default setting (which is inferred if no `except`, `only` or `rules` key is present) for jobs is `only: [branches, tags]`, according to the documentation [27]. Since `branches` implies `schedules`, a fair amount of correctly working pipelines can be scheduled without further changes to the configuration.

However, GitLab will only schedule pipelines with correct configurations. A misconfigured pipeline (passing linting, but invalid at runtime, e.g. because of prerequisite jobs not being instantiated) will not be scheduled, and no error message is given, nor any other indication that the pipeline was not scheduled or why it was not scheduled. Thus, this breakage will go *completely unnoticed*. The scheduled pipelines will simply not run at the scheduled time. If the user triggers the scheduled pipeline manually, they will get an indication by the platform that the pipeline has been successfully started - but no run of the pipeline will take place and the user is never been informed that the run hasn't happened.

This makes adding jobs to any pipeline that used conditions an error-prone task, but pipelines which use scheduling have an exceptional high error-potential due to the non-existing error message and misleading behavior of the platform.

In the following, I will refer to the combination of `except` and `only` as well as `rules` as *conditions*. A condition is said to be *weaker* than another condition if a job is instantiated in at least all of the cases or *more* cases than the other job. *Stronger* is used analogous. Conditions can not be compared if they lead to the instantiating of jobs in completely different circumstances.

This leads to the following, dual description of the requirements for conditions in a valid pipeline configuration:

- A job preceding another job must have a *weaker* or equal condition as the job succeeding it.

- A job succeeding another job must have a *stronger* or equal condition as the job preceding it.

Since the DSL can be thought of as describing *intent*, there are multiple ways this can be dealt with when deriving a configuration from the model:

1. Semantically check that the requirements are met, emitting an error if they are provably not or a warning if their equivalency can not be established.
2. Forward propagation of conditions through the graph, restricting execution of successors.
3. Back propagation of conditions through the graph and relaxing the conditions of preceding jobs.

All of these are attainable goals, with different strengths and weaknesses.

When only the simple forms of **except/only** are used, the check is either a subset test when using literal values, or an equivalency check for regular expressions. In case rules are used or the more complex configurations of **except** and **only**, the checks amount to significantly more complex propositional logic equivalence tests.

While only making an author aware of the problem is a usable approach, this still leads to a lot of duplication when authoring a pipeline and require the author to manually distribute the conditions to all jobs. This is not as laborious as it sounds at first, since the same condition (an **Except**, **Only** or **Rules** block) can be connected via edges to multiple jobs, and one job accepts multiple conditions that are merged during code generation.

But it still seems preferable to further reduce manual labour and to automatically ensure that the correct conditions are set for any job, based on the path taken through the graph.

There are two possible ways to interpret the intent an author might have had in the DSL, and it is not immediately obvious which interpretation is the more intuitive one.

Adding a job to an existing pipeline that restricts execution of jobs to non-default conditions might lead to a violation of the requirements laid out above, either when the job is added as successor of a job which has more narrow conditions, or if it is added as predecessor of a job which has a wider set of conditions.

Since it is reasonable that a job which is added behind some other job should only be executed when the job preceding it has also been executed, it seems reasonable at first to propagate conditions from front to back, thereby narrowing the conditions under which a job is executed, the deeper in the path taken through the graph it is encountered.

However, extending such a workflow might have unintended consequences when the developer adding a job deep in the graph, unaware of the restrictions that come before. Since

the default on GitLab is to execute jobs exactly under the circumstances described for the job itself, a developer might reasonably assume this to hold true as well.

On the other hand, widening conditions in earlier jobs might also be undesirable. If a job introduces new conditions under which it is to be executed, environment variables needed for earlier jobs might not be set. For example, an earlier job might depend on the tag name. Adding a later job that relaxes the conditions might break the pipeline, because an earlier job is now executed without the necessary variables set.

The third option is to do nothing and only warn the user about potential problems. Further user testing and feedback is needed to settle on a final strategy, but all three ways are possible and reasonable solutions.

6.2 Generalization

While this thesis has shown that visually modelling CI/CD workflows for a specific platform – in this case, GitLab – is feasible, the vast potential for generalization has not yet been realized.

There are seven high-level concerns that can be identified by comparing the feature set of available solutions.

1. Build script
2. Artifact Production and Consumption
3. Caching
4. Environment
5. Services
6. Conditions
7. Dependencies and Ordering

Build script The main component of CI/CD pipelines are build scripts. Often divided in three parts, a main script, a script to execute before the main script (sometimes referred to as setup or *prologue*) and a script to execute after the main script (sometimes referred to as teardown or *epilogue*).

Some platforms allow prologue and epilogue to be re-used for multiple jobs (e.g. SemaphoreCI), making them useful to provide a properly configured build environment and to run common checks on multiple jobs.

Since the three scripts can easily be appended and combined into one script, a DSL that supports these three steps can also target platforms that choose to only support a single build script.

Artifact Production and Consumption Another aspect of CI/CD pipelines is the production of artifacts by a job. This may include the actual product, e.g. as a binary, but might also include documentation and test reports.

A key aspect of artifacts is configuring which artifacts are produced by which job, and passed to another job for consumption. Some platforms allow implicit artifact passing (GitLab), others handle artifact passing very explicitly.

In this regard, passing of artifacts can be thought of as a data flow.

By making artifact passing explicit within the generalized DSL, it is possible to target both platforms that handle artifacts implicitly and those that require it explicitly.

Caching Caching of dependencies e.g. Node.js modules, NuGet packages and Maven artifacts can greatly speed up the build process, but some intermediate results might also be cached between builds to prevent re-building the whole project from scratch each time. As such, all platforms offer caching mechanisms to allow the developer to specify directories or files that are to be cached between pipeline runs.

While the details of this implementation varies, it is generally similar enough to be described in an abstract way.

Environment The environment in which the build script is executed needs to offer all tools required by the build script itself, e.g. the .Net SDK, a JDK and Maven, or Node.js and `npm`.

While it is possible to run on an empty platform, only choosing the base OS, and installing the complete tooling every time, this is very time-consuming and inefficient. To this end, docker has become the de-facto standard in pipeline virtualization. Users can either choose pre-defined docker container (e.g `node:12-alpine`) or create their own, custom-tailored container with all the needed tooling, and use that.

Thus, a DSL that allows a docker container to be specified and configured can easily be used with a wide variety of target platforms.

Services Similar to the above, a job might require additional services to be available. As an example, integration tests might need to have a database available for testing. While it is again possible to install this service onto the build platform each time the job is run, it is far easier to describe a service as a docker image that has to be available, e.g. `mysql:latest`. This has become the prevalent solution to specify required services, and is supported by most CI/CD solutions.

A generic way to describe services via docker images can thus target a wide variety of platforms.

Conditions Each platform offers a way to specify when – under which circumstances or conditions – each job shall be executed. The specific mechanisms vary wildly between platforms. Some, like GitLab, offer explicit support to *exclude* a job from a pipeline or to *include* it, as well a generic rule-based approach [27]. SemaphoreCI only offers so called "*skip conditions*" [43], which *exclude* a job from a pipeline. GitHub Actions support triggering workflows via events [24], *including* a job only when the specified conditions are met. The exact feature set varies a lot, as does the specific way in which these conditions are to be specified.

At the same time, this is where the most potential for generalization lies. Common concepts for CI/CD pipelines can easily be identified. For example, it is typical to include or exclude jobs based on branch names or tag names, or simply by the fact that they were triggered by tag creation. The latter is a common approach to creating releases, by having jobs that create the release only be run when a tag is created, while the other jobs run on every commit.

It is also where visual authoring can greatly benefit the author. Most platforms offer very little capabilities to help users generate correct or sensible configurations.

For example, it might seem sensible to use the *skip condition* `branch == 'main'` for a job when using SemaphoreCI. At first glance, this should restrict the job to only be executed on the `main` branch. The visual editor provided by SemaphoreCI will accept this input and not warn the user that this is, in fact, syntactically invalid. After committing the erroneous workflow to the repository, the pipeline will fail with the error message `Error: "Invalid 'when' condition on path '#/blocks/0/skip/when': Syntax error on line 1. - Invalid expression on the left of '=' operator."`. The reason is SemaphoreCI's non-idiomatic use of `"=`" for the equality test instead of using the more common `"=="`.

Rig already includes a grammar that aids the user in writing correct condition expressions, but this still leaves room for improvement. Since variables can be arbitrary, the grammar has to support arbitrary values as variable names. While support for predefined variables is already available, vigorous validation of custom variables is not yet implemented, since

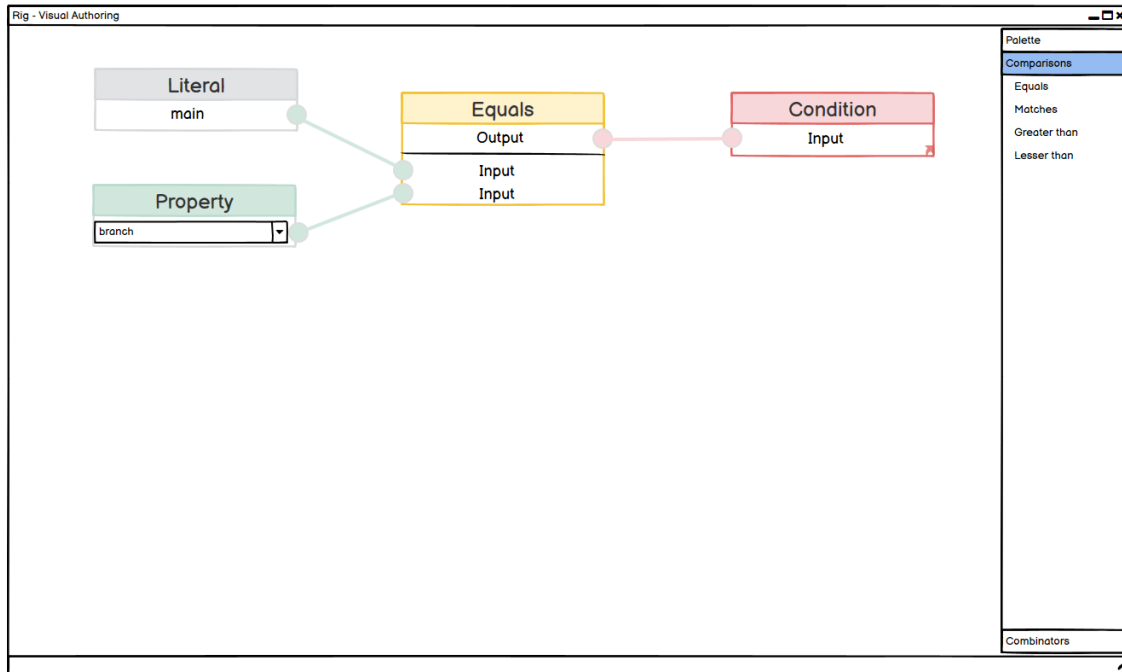


Figure 6.3: Mockup of visual authoring for conditions

those have to be collected from the DSL, and then injected into the grammar parser as valid values at runtime. As shown in the following, this can instead be achieved by employing a more model-driven approach to conditions as well.

Conditions offer an excellent opportunity to employ visual authoring, something not yet done in Rig. Both Blenders Shader Nodes (cf. Figure 2.1), but to an even even greater extend, Unreal Engines Blueprint Editor (cf. Figure 2.2), can greatly serve as inspiration for such an endeavour.

Figure 6.3 shows a mockup of visual authoring for conditions. The green block, labelled *Property*, represents some external property, entity or some aspect of the environment or pipeline. In this case, the chosen property is the branch on which the pipeline is to be run. The grey block represents a *literal* value. Both of these blocks have a single output each, which are connected to the inputs of the yellow *equals* block, which represents the binary operator of the same name.

The single output of this block is connected to the final *output* block, which makes the condition available in the pipeline. Figure 6.4 show how this block is connected to the job in the pipeline definition. By employing visual authoring in this manner, users are automatically guided to create only valid configurations. The configuration does not exist in text form, but is already represented as structured data and can readily be consumed by tools operating on abstract syntax trees (ASTs) for further processing, e.g. semantic equivalence checking, and does not need to be parsed again.

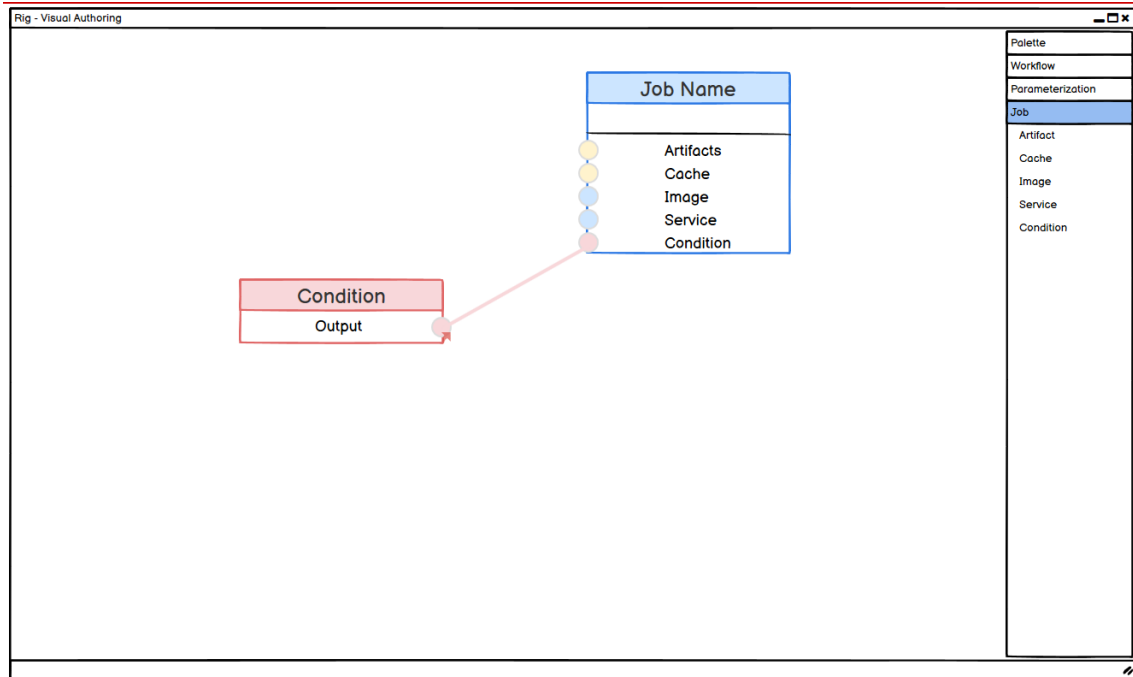


Figure 6.4: Mockup of the condition embedded in the CI/CD workflow. Clicking the condition will enable the user to author the condition as shown in Figure 6.3

By abstracting conditions this way, they are actually made more powerful, not less. Even when only targeting one platform, the increase in usability is huge and makes authoring and subsequent generation of correct configurations much easier.

The condition shown above can be exported both to SemaphoreCI, GitHub Actions and GitLab, and possibly other platforms. The equivalent code for these platforms can be found in Listings 2 for SemaphoreCI, 3 for GitHub Actions and 4 & 5 for GitLab.

Thus, a more generalized description of conditions both enables better and more intuitive visual authoring, can serve as better foundation for further processing, and can support the targeting of a wide variety of target platforms, and is thus the most important and impactful improvement that can be made to the current DSL.

Dependencies and Ordering The above paragraphs lay out the aspects of a single job. By composing jobs together, they form a pipeline. For this, it is important to be able to define the order in which jobs are to be executed. This can be modeled as a directed graph, and such a model has already been given in [46], which was the foundation of this thesis. The current DSL supports this well and easily supports emitting correct configurations wrt. job ordering for a wide variety of platforms.

```
1 skip:
2   when: branch != 'main'
```

Listing 2: SemaphoreCI condition with branch name equality test

```
1 on:
2   push:
3     branches:
4       - main
```

Listing 3: GitHub Action with branch name equality test

```
1 rules:
2   - if: '$CI_COMMIT_BRANCH == "main"'
```

Listing 4: GitLab branch name equality test using rules

```
1 only:
2   refs:
3     - main
4     - branches
```

Listing 5: GitLab branch name equality test using only

6.3 Visual Authoring for Regular Expressions

Some properties in the DSL take regular expressions as values. In the case of GitLab, regular expressions are implemented using RE2 [27]. Currently, Rig allows free-form text entry for regular expressions and needs to perform linting to ensure correctness of the input value.

Regular expressions suffer from many of the same problems as YAML files. Their syntax is complex and arcane, re-use is limited at best and large expressions are hard to read for humans.

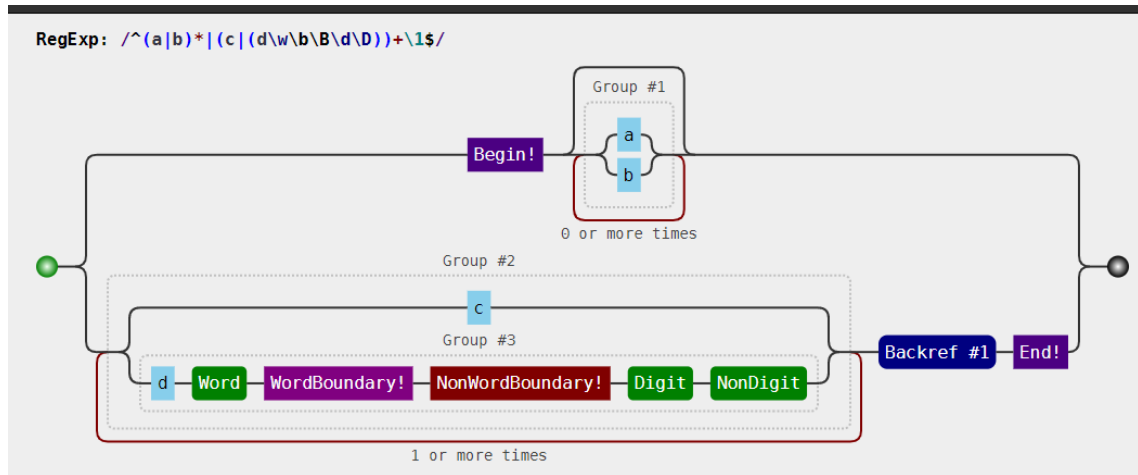


Figure 6.5: Visualization of a regular expression using Regulex

In order to ease writing of regular expressions, visualizations have been developed, for example Debuggex¹ and Regulex². These allow the developer to visually inspect their expression to better understand it and construct correct expressions more easily. Figure 6.5 shows a visualization of a regular expression using Regulex.

Current solutions focus on merely *visualizing* regular expressions. Since regular expressions can freely be converted to and from DFAs, it is imaginable to write a regular expression editor in which regular expressions can be built visually, via *drag and drop*.

It stands to reason that such an editor can be build using CINCO, and subsequently be integrated into Rig as editor for regular expressions, furthering the visual authoring aspect and moving further and further away from textual input.

A regular expression built with such an editor should be correct by design, and must not need further linting.

6.4 Reducing Edge Count via Target Extension

The DSL suffers – to a certain degree – from edge proliferation. With an increasing number of targets which parameterize a job, the number of edges that connect back to a job scales linearly. Figure 2.7 has already been discussed as the model-driven basis for this DSL. In that figure, this kind of edge proliferation can already be seen. Looking at the *"Deploy"* job and the `TARGET_URL` parameter, one can see that it has no less than four incoming edges, one for each target. The same is true for the `API_URL` parameter of the *"Build_SPA_Client"* job.

¹<https://www.debuggex.com/>

²<https://jex.im/regulex/>

The DSL presented in this thesis allows for a much wider range of parameterization options, since all of the properties can be parameterized. Making heavy use of the parameterization capabilities can easily lead to the explosion of the edge count, rendering the DSL both hard to visually comprehend and difficult to work with.

Adding a new target requires $N + 1$ edges – one connecting the last job to the target, and one for each parameter. This makes adding new targets an increasingly difficult task as the numbers of parameters and already existing targets increase, since the edges start to overlap and cross on their way through the model.

The model-driven approach given in [46] already includes an ordering mechanism for targets. It has not been implemented into this DSL since it did not offer a useful feature as it was. Reintroducing such an ordering mechanism, but instead of just having an ordering semantic with an extension semantic would provide various benefits.

A proper extension mechanism, modeled after UML, might drastically reduce the edge count.

When connecting a target to an existing one via an extension edge, the new target has to provide the same parameters as the existing target. There are two possible ways to correlate the parameters of the new target with the parameters of the old target.

A possible solution is via naming. Parameters could have names, and a new target would need to provide parameters with the same names as the old target. This solution has two disadvantages. Parameters need to be named even if the target is never extended, adding additional work for the author. Furthermore, names are just strings and thus prone to typing errors. While proper tooling can help via auto-complete and code-assist as well as checks during authoring, this solution is nevertheless brittle.

The other way to correlate the parameters is via edges from the parameters of the new target to the corresponding parameter of the existing target. It may sound counter-intuitive to combat edge count proliferation with more edges, but there is a significant difference between both approaches. The edges connecting the new target to the old target are much more constricted in area and are local. The edges going from a target to the jobs might take a long path through the whole model. It is a high amount of long edges that makes the DSL unreadable. Short, local edges do not hinder comprehensibility of the DSL in the same way.

They are also easier to work with as an author. Since connecting edges from a new target to an existing target is usually restricted to a small area of interest (if the author chooses to cluster their targets), it is easier for authors to keep track of what they are doing or where their edges need to be connected to. This is in stark contrast to having to connect edges through the whole model, maybe even back to the very first job.

Thus, adding an extension parameter that does not rely on brittle parameter names, but instead uses the strong guarantees that properly modeled edges offer seems preferable.

Adding such a mechanism would also greatly improve re-usability. A pipeline can more easily be reused by adding new targets. Chapter 6.5 discusses this in more detail, especially 6.5.2.

6.5 Towards Reusability

Reusability was a big concern raised in the introduction and motivation for this thesis. While it was a motivation for the thesis, many features that would improve reusability have not yet been implemented into Rig, although the foundation has been laid. The following sections lay out approaches to facilitate reuse of already written and tested pipelines.

6.5.1 Prime References

CINCO implements so-called *prime references*, which can be used to reference elements from other or the same grammar, across multiple files. By introducing prime references into Rig, pipelines may re-use jobs defined in other pipelines. Its also thinkable to allow prime references for other elements. Also, extending whole pipelines and allowing sub-pipelines in Rig could greatly enhance reusability. Investigation into which elements can be used in which way has still to be done, but the potential is huge. Prime references would also be the basis on which a "*marketplace*" as discussed in the next section could be built.

6.5.2 Abstract Targets

The basis for abstract targets is a target extension mechanism as discussed in Section 6.4. Abstract targets would allow a user to model a workflow without instantiating any job, only providing the "*skeleton*". By adding concrete targets via the target extension mechanism, the user could then generate the actual workflow. This would allow users to create and maintain their own library of readily available workflows, especially in combination with prime references. Creating the workflow for a cloud-based SPA could be as easy as importing the existing workflow, which includes the abstract target, and adding a single concrete target that parameterizes the abstract target.

This would also greatly enhance the opportunities to share a workflow with other developers e.g. via a common marketplace as discussed below.

6.5.3 Sharing Definitions via Marketplace

Workflows often share many common aspects. Existing CI/CD solution often offer templates to get developers started with their workflow, including GitLab [25], GitHub [22] and SemaphoreCI [41]. Users will then be able to refine those templates. Adding a similar capability to Rig would increase re-usability of the DSL.

However, with prime references and abstract targets, Rig is able to offer more in terms of re-usability than simple templates that are then changed and adapted. By utilizing these features in concert, a user could reference the abstract target of a workflow from the marketplace and then just parameterize it with their own parameters, without the need to adapt or change the original workflow. This would also allow improvements made to the base workflow to automatically affect the workflow that extends the abstract target.

In order to avoid breaking workflows, a versioning mechanism for prime references would also be needed, so that author can employ version pinning to fix the version of their workflow, only deliberately updating to new versions.

6.6 Rig as Integrated Development Environment (IDE)

While a major goal of the DSL is to facilitate builds via CI/CD pipelines, local building still plays an important role in day-to-day software development. Much work has gone into hot-reloading, e.g. Quarkus (via `mvn quarkus:dev` [39]) or Webpack (via `webpack start` [44]), so that developers can get instant feedback on their work.

When builds get more complex, it is often necessary to provide build scripts to enable developers to build their code locally. This leads to duplicate effort, when both the CI/CD pipeline configuration and a single build script have to be maintained.

These issues can be tackled by Rig; since a full description of the CI/CD pipeline is available, executing the pipeline by various means and monitoring the status seems attainable. That way, maintaining separate build scripts for local building becomes superfluous.

Generating a build script The code generator can emit a build script for a single target, or multiple scripts for multiple targets, that can be run locally, simply by concatenating all scripts (and prologue and epilogue scripts, if applicable) in the correct order. Where the CI/CD pipeline leverages parallelism, this might be lost in the process, but a correct linear ordering of jobs can be derived from the graph. Thus, Rig could be used to help create and maintain a build script.

By adding marker statements to the process, Rig could then monitor the output of the generated script and offer continuous, real-time feedback by marking jobs as finished, in

progress, or failed. Rig could also forego emitting a single build script, and instead emit the necessary commands on an as-needed basis while the build progresses.

This poses some serious restrictions, since the environment as specified in the DSL (both wrt. the docker image as well as the services) can not be loaded, but must be installed on or be provided by the developers machine.

Emulating the pipeline locally Some platforms have (limited) support for local execution of pipelines. GitLab offers a runner, which can be used locally, albeit with limitations. It has been used successfully to run pipelines locally [48]. Since the DSL is currently tied to GitLab and written specifically to support it, it initially seemed to be worthwhile to investigate the possibility of adding support for local execution of pipelines in this way.

There are various open problems with this approach e.g. the population of variables and the simulation of certain conditions, e.g. tagging. Other tools like `glci`³ have been built on top of the runner to combat these issues and make local execution of pipelines easier.

Since the move to generalize the results of this thesis, this approach has fallen out of favor of the author. Instead, generalizing the DSL offers the possibility to employ a more powerful model as explained below.

Orchestrating a build Gantry⁴ is a toolkit to orchestrate docker containers from Java. If the host machine offers has an active Docker service, Rig could be used to start, stop and destroy containers and services as described by the DSL, and to run the build scripts on them. By monitoring the containers output and status, Rig can offer instant visual feedback as the pipeline progresses.

Since everything is under control of Rig, simulating the pipeline under various conditions is feasible. Thus, unit-testing and integration-testing the pipeline itself becomes a possibility, a feature not yet seen in any other solution. Caching errors, dependency errors and many other classes of errors previously only detected at runtime (in live pipelines) can thus be caught long before they arise, and can be dealt with preemptively. This would present a major step forward towards error-free CI/CD pipelines and towards proper authoring and craftsmanship, instead of relying on trial and error as main method for the derivation of correctly working pipelines.

³<https://github.com/mdubourg001/glci>

⁴<https://gitlab.com/scce/gantry>

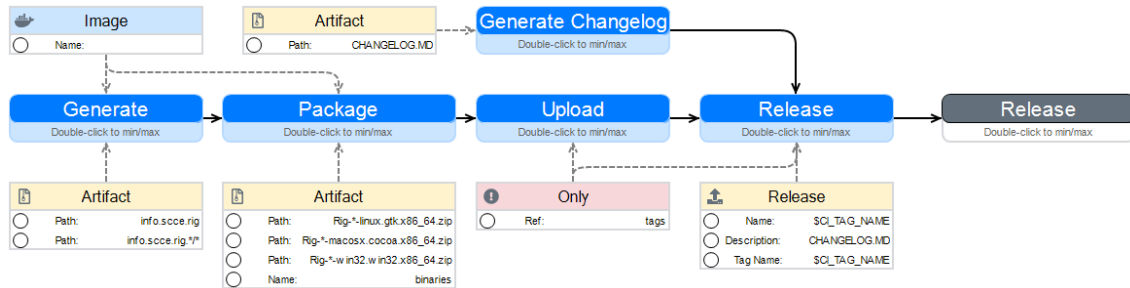


Figure 6.6: Build process of Rig, modelled in the DSL.

6.7 Bootstrapping Rig

Implementing DevOps in the development of Rig remains an open task, although in principle, Rig supports all features needed to describe its own CI/CD pipeline, as shown in Figure 6.6.

Building Rig involves multiple steps:

- At first, the CINCO product must be generated. This is done by opening the project in Eclipse, and then choosing the `Generate CINCO Product` menu entry. This is represented by the *Generate* job.
- Afterwards, the proper Maven artifacts have to be generated. This is done by the job named *Package*. Since CINCO product generation creates a clean state first, by removing all target directories, the required Maven artifacts have to be restored – for example from another branch. Then, the required CINCO artifacts needed for building the product have to be generated. In the version presented here, CINCO artifacts are not cached between builds, but this would also be a possibility to speed up build times. Afterwards, the project can be build, either via `mvn package` or, as alluded to by the job title, with `mvn verify`, which includes not only building the package, but also execution of unit- and integration tests, if configured. Building Rig via Maven produces executables for three platforms: Windows, MacOS and Linux. By including these binaries in the `artifacts` property, they are stored by GitLab as artifacts produced by the pipeline.
- At the same time, a changelog can be generated. By following the *Conventional Commits*⁵ specification, this process can be automated.
- If and only if a new tag is added, the *Upload* job is responsible for uploading the binaries as so called "*Generic Packages*" to the GitLab package registry, where they are available for download to the general public.

⁵<https://www.conventionalcommits.org/en/v1.0.0/>

- Similarly, the *Release* job adds a proper release to the GitLab repository for every tag. The release is named the same as the tag, and includes the generated changelog as description and the binaries uploaded to the package registry as downloadable artifacts.
- Since Maven already takes care of producing artifacts for the three target platforms, there is no need for multiple targets. A description of the build process with multiple targets is also a reasonable option, with one target specifically for tags, and one target that is only used to produce the binaries. However, since the jobs uploading the binaries to the package registry and creating a release can be restricted to tags only, adding another build target would only increase the complexity of the pipeline description, while adding no new functionality. Hence the pipeline has been expressed with one target only.

The process as shown above remains manual labour, for the moment. All but one of these jobs can already be automated. But the very first job shown in the pipeline – the generation of a CINCO product – can not be automated and must be done manually, by opening the project in Eclipse and triggering the product generation via mouse-click. All of the other jobs can be automated and with shell scripts and automatically be executed as part of a CI/CD pipeline.

Since command-line execution is required for CI/CD pipelines, it is as of now not possible to automate the build process for Rig. The addition of CLI support to CINCO would empower developers to adopt the DevOps lifecycle for CINCO products. Adding support for command-line execution to an existing Eclipse RCP application has previously been investigated and been described in [17].

This highlights both a weakness and a strength of the DSL. The DSL can never offer more than the underlying tooling. The requirement to manually generate the CINCO product can not be waived by the DSL. But on the other hand, the DSL can still be used as a documentation tool to *describe* the build process, until it can finally be implemented as working, executable CI/CD pipeline.

Chapter 7

Discussion

This chapter discusses the DSL and results obtained in Chapter 5. A comparison of the DSL presented herein with another popular CI/CD tool that offers at least limited visual authoring capabilities, namely SemaphoreCI, is given in Section 7.2.

7.1 Edge Proliferation

With the large amount of parameterization possible in the DSL, a great danger is visually cluttering the canvas with a large amount of edges, making the workflow visually hard to comprehend.

The DSL already offers some ways to mitigate edge proliferation, but they have to be employed by the author. If jobs leading to a target share a common Docker image, this image can be configured on the target. The same holds true for services and the condition elements (`rules`, `except`, `only`). By setting these properties on a target, they automatically get propagated through the graph, and the author does not need to manually link these properties with edges to every job.

The problem can be lessened even more by allowing targets to extend already existing targets. This mechanism has been described in Section 6.4 and is on the roadmap for future versions of Rig.

DSLs that focus on only describing jobs and their relation to each other do not suffer from the problem in the same way, but they also display a lot less information in-canvas. Balancing visual authoring capabilities by moving the authoring onto the canvas and thus providing the information and configuration in an easily accessible way has to be properly balanced against visually cluttering the screen.

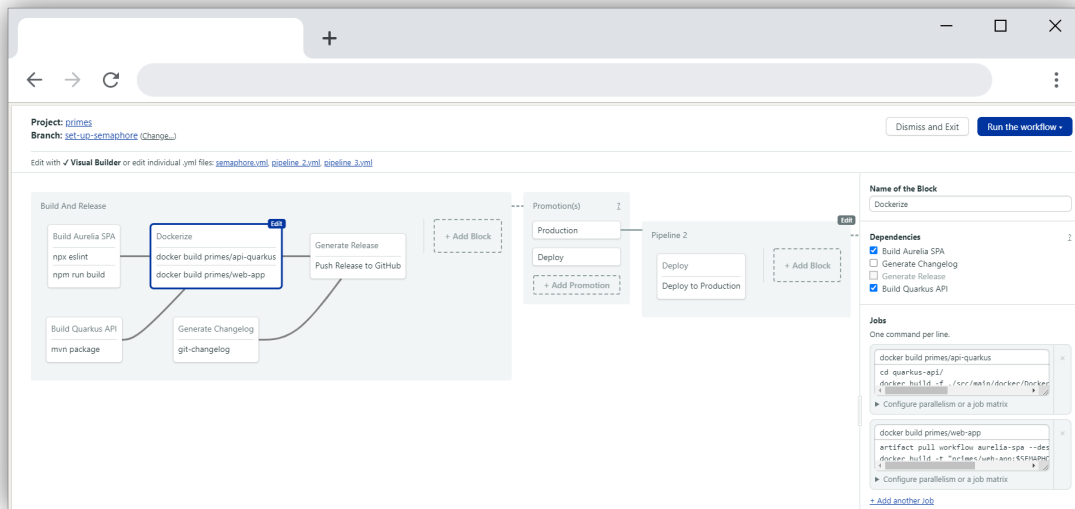


Figure 7.1: A cloud-based web application deployed via Docker, using Aurelia as SPA for the frontend that consumes a web API backend implemented via Quarkus. Modeled here with SemaphoreCI. Compare with Figure 5.1.

With the implementation of extensible targets and the generalization of the language, it should be possible to create a fairly concise, but still very informative DSL that offers an intuitive way to visually author a workflow.

7.2 Comparison with SemaphoreCI

SemaphoreCI has been chosen as target for a more in-depth comparison with Rig due to its visual authoring capabilities, which are even more limited with the other tools compared in Table 1.1. To this end, a workflow that largely resembles the case study for Rig, as presented in Section 5.1, has been modeled in SemaphoreCI and is shown in Figure 7.1.

The visual editor of SemaphoreCI visualizes a workflow, but does not allow visual authoring. The elements can not be interacted with on the canvas, except for selecting them. Deleting selected elements by pressing the `DEL` key is not supported, although such an interaction is common in almost any other software that offers selectable elements. Instead, an author has to find the correct button in the configuration panel of the block.

Instead of using jobs as their main modeling element, SemaphoreCI uses so called "blocks". A block can contain multiple jobs, which are then executed in parallel. Blocks are scheduled according to their ordering in the dependency graph. Figure 7.1 shows how SemaphoreCI visualizes the graph.

Instead of targets like the proposed DSL, SemaphoreCI uses so called "*Promotions*", which are described as suitable for "deploying to production on master builds or deploying to

a pre-production environment on topic branches" [42]. Promotions are executed either manually by the user or automatically when certain user-supplied conditions are met. A *promotion* then executes another pipeline as specified in the configuration.

This approach notably lacks the ability to parameterize builds. If a topic branch should use a different configuration – e.g. different credentials – then one can not parameterize the build pipeline. Instead, one has to duplicate the existing build pipeline and then change the required parameters.

Dependencies are specified using checkboxes when a block is selected, as shown in Figure 7.1. The blocks are then arranged automatically on the canvas. Authors does not have the opportunity to sort the blocks themselves. Thus, they are unable to make the workflow visually clearer in situations where the automated layout produces undesirable results, which is a possibility in highly interconnected build pipelines.

In summary, SemaphoreCI uses four elements to visualize the build process: blocks as containers for jobs, the edges that connect blocks to visualize dependencies, the pipeline as container for blocks and promotions which trigger subsequent pipelines. None of these can be altered visually, only via configuration panels.

SemaphoreCI does not model anything else – parameterization of jobs is notably absent from their visualization. Artifact passing has to be done explicitly via CLI scripts, instead of being derived from the visual model. The same applies to caching, conditions and other model elements. The absence of these elements from the visualization also means that the visualization only offers a superficial view of the build process, making inspecting the contents of the configuration panels a necessity in order to understand the build process at all.

SemaphoreCI also lacks proper validation of its inputs. The free-form text fields are not validated in any way, not even syntactically. The tool does not offer the opportunity to test a given configuration. Instead, configurations have to be committed into the target repository. When setting up SemaphoreCI for the first time, the tool chooses the `setup-semaphore` branch as target for its commits, mitigating the impact on existing production branches. However, it is still necessary to commit the workflow to the repository in order to get any feedback if the configuration is valid. Syntax errors are thus only detected when the configuration is already part of the repositories history. An example for this behavior has briefly been discussed in Section 6.2 (pg. 41). The inability to test pipelines makes it possible that errors in rarely used parts of the pipeline stay undetected for a significant time and are only detected when the needed part of the pipeline fails, possibly at very inopportune moments.

In comparison, the proposed DSL offers a much wider range of visual elements, a much richer interaction scheme and much more comprehensive (although not perfect) syntactical and semantic checks, with a huge potential for future improvements as shown in Chapter 6.

The complete palette of the proposed DSL not only encompasses jobs and targets (connected via edges) as main modelling elements for the graph, but also offers visual elements for all other configuration options. Thus, a pipeline can be understood only by looking at the visualization, without diving deep into the property panels. The elements can be freely arranged by the author, allowing them to topically group elements.

Parameterization of jobs via targets makes it possible to add new deployments or build targets like new architectures easily, without completely duplicating the pipeline.

Rig implements XText¹ grammars for most of its free-form input fields and applies custom validation on a best-effort basis. Unfortunately, tying the tool directly to GitLab means that the tool has to directly support all quirks and idiosyncrasies of the target platform. Generalizing the tool as described in Chapter 6 allows for a more powerful DSL and tool, and also allows one to hide the idiosyncrasies of the target platforms from the end user.

In summary, the DSL presented in this thesis offers a much wider range of elements and the tool generated by CINCO (dubbed Rig) allows for richer interaction with the DSL than the visualization of SemaphoreCI.

7.3 Comparison with GitLab’s Visual Pipeline Editor

The plans of the GitLab maintainers and contributors are similar, but not as far reaching as this work. Over the past months, they have succeeded in implementing a non-interactive visualization of their pipelines (cf. Figure 2.5), but the work towards visual authoring is still ongoing. Their 3-year plan includes some limited visual authoring capabilities [45]. Notably absent from their approach is the concept of targets, and while they are focusing on displaying the relationships between jobs as edges (implemented in YAML via the `needs` keyword), their plans lack the inclusion of other visual elements to the canvas. Since they do not (as of now) plan to include more visual elements, their approach lacks the opportunities of generalization that this DSL offers. Which is understandable, since they only target their own platform.

7.4 CINCO as Workbench

As described in Chapter 3, the DSL was implemented using the meta-modelling tool CINCO. Using CINCO allowed for the quick building of a working tool. Overall, the

¹<https://www.eclipse.org/Xtext/>

usage of CINCO proved very effective in achieving most of the goals laid out in the motivation of this thesis. CINCO is still in active development, with Version 2.0 having been released on March 15th, 2021 [7]. While employing CINCO has had great advantages, a few areas of improvement could be identified.

Most notably, CINCO does not support command-line execution of its product generation routine. CINCO products have always to be built via manual interaction with a menu inside the application. This means that adopting the DevOps lifecycle for the development of Rig is – at least for the moment – impossible, since a CINCO product cannot be built in a CI/CD pipeline Section 6.7. The irony of proposing to improve CI/CD configuration authoring and easing the adoption of DevOps while not being able to do so themselves was not lost to this author.

Since CINCO generates a full Eclipse RCP and does not support incremental builds, build times are slow. The machine employed to develop the tool in this thesis needed between two and four minutes to build the CINCO product. This made rapid iterations of the language difficult. Adopting test-driven design with lots of small, incremental changes to the DSL proved to be exhausting. The long build times give an incentive to pool changes and only build the product when enough changes have been polled. This makes finding a change that introduces an error difficult.

The heavy coupling of the style language and the graph language means that even superficial visual changes have to go through a full build of the CINCO product. Since visual changes often necessitate visual inspection of the results and ensuring they work out as expected, often with small changes having to be done until the desired look is achieved, it proved to be very time-consuming to properly style the DSL.

In Chapter 2, multiple other visual authoring tools have been presented. A component found in multiple of these solution is in-canvas editing (especially in Blenders Shader Node Editor and Unreal Engines Blueprint Editor), the ability to directly interact with attribute values of model elements. CINCO supports displaying attribute values via `text` style elements. Support for interactive elements like textfields (for single line text) textareas (for multi-line text), dropdowns (for enum-like values), sliders or spinners (for number inputs), combo boxes (for free-form text with some preconfigured options, as needed in Rig for *conditions*) and checkboxes (for boolean inputs) would greatly enhance user experience in this regard.

Due to being an Eclipse RCP, CINCO products are rather heavyweight. For an MGL just shy of 1k LOC, CINCO produces an executable of the size of 350MB *per target platform*. The total size of all three binaries is just over 1GB. Startup of the tool is rather long due to having to load a rather large collection of packages. Furthermore, the tight integration

with Eclipse might hinder adoption by developers who prefer other IDEs (notably IntelliJ or VSCode).

On the other hand, this also demonstrates the great capabilities of CINCO. The core code for Rig is about 1k LOC in the MGL, about 700 LOC for the style file and again about 1k LOC for the code generator. This rather small model is enough to produce a fully fledged Eclipse RCP with a full visual editor. It wouldn't have been possible to create such an extensive tool without deploying a meta-modelling framework like CINCO.

Chapter 8

Conclusion

As shown in Chapter 5 (and, to a lesser extend, Section 6.7), the DSL is capable of expressing complex, real-world CI/CD processes and to generate to correct configuration files from them.

By providing a clear iconography and color scheme, workflows remain readable even when they become more complex, and can thus serve as documentation and communication tool as well as easing adoption of the CI/CD process and DevOps methodology.

By specializing the DSL for a single target platform, code generation from the DSL is rather straight-forward. Simple transformation rules can be followed in a fixed order by the code generation algorithm. The code generation strategy as shown in Chapter 4 has fewer lines of code than the DSL itself.

While the currently implemented tool only guarantees syntactically correct workflows to be generated, the DSL as designed is expressive enough to express the authors *intent*, allowing semantic checks to be performed or code generation to become more complex to ensure that only reasonable configurations are emitted. Much of this has been described in Chapter 6.

CINCO is a very capable platform to create graphical DSL and visual authoring tools. However, some limitations still exists. Many values in the DSL can not be manipulated directly on the canvas, unlike in Blender, but have to be manipulated via a separate panel. In-place editing capabilities would greatly improve the CINCO framework in this regard.

Work on this thesis also uncovered gaps in the current practices wrt. the testing of CI/CD pipelines. Current approaches focus on making editing CI/CD configurations easier by applying static checks such as linting, configuration validation and visualization or, as presented in this thesis, a complete visual authoring tool. However, creating a framework for unit/integration tests of the pipelines themselves is a currently largely unexplored area of research.

This thesis produced a working visual authoring tool for CI/CD pipelines for a single target platform, GitLab, and laid out a foundation for future work in this area, especially wrt. generalization and further improvements to visual authoring instead of relying on textual inputs.

The tool has been demonstrated to work with a case study (Section 5.1), and can greatly reduce the barrier of entry for new users. Features are easily discoverable through drop-down menus and a drag-and-drop palette.

Targeting GitLab as a single platform seemed reasonable when this thesis was conceptualized. It appeared to be the next step from a model-driven approach toward a usable implementation. Research and reviews of existing solutions conducted during this thesis has revealed that there is a huge potential in generalizing this problem (cf. Section 6.2). In hindsight, targeting a single platform was a mistake that only limits the power and expressiveness of the DSL, not enhancing it. Making the DSL specific to GitLab means that code generation and all semantic checks have to support all quirks and idiosyncrasies of the platform. A more high-level DSL can opt to describe the intent of the author on a much higher level, can do semantic checks more reasonably and can then emit the specific code for the target platform. Future work is needed to explore generalization of the DSL further and to implement code generation strategies for multiple platforms. It will also greatly enhance visual authoring capabilities for the current target platform, GitLab.

This work also revealed the huge potential of generalizing CI/CD process overall. Many of the same concepts can be found at various CI/CD providers and in many implementations, but the concrete syntax to describe essentially the same workflow and concepts is vastly different, even though in most cases, YAML is employed. Thus, knowledge of CI/CD processes is not easily transferable between vendors, requiring re-writing the configuration, with extensive work needed to familiarize oneself with the target platform.

A standardized specification could alleviate many of these problems and concerns. By describing the CI/CD process in a standardized way, IDE vendors could offer support for authoring of such configurations. Switching vendors would become a lot easier, reducing the danger of vendor lock-in. There already is limited interoperability between CI/CD solutions, for example by using GitLab CI/CD for GitHub repositories [29]. By having a standardized, platform-agnostic description of the CI/CD workflow, developers can concentrate on their workflow, not how to express it for the platform they happen to use. Such a standardized description could also serve as basis for tools that locally build, test and integrate the product a developer works on.

Such a specification could also serve as target for a new, generalized visual DSL that helps developers to visually express their workflow, and could thus become a main driver for easier adoption of the DevOps methodology.

Appendix A

Further Information

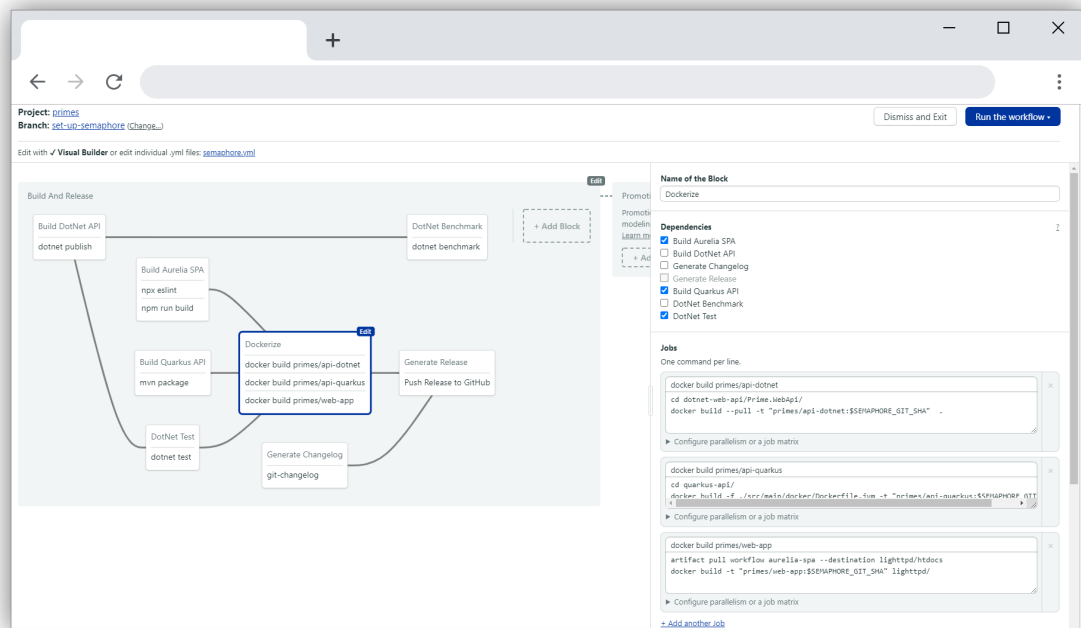


Figure A.1: Example workflow of a cloud-based web application in SemaphoreCI

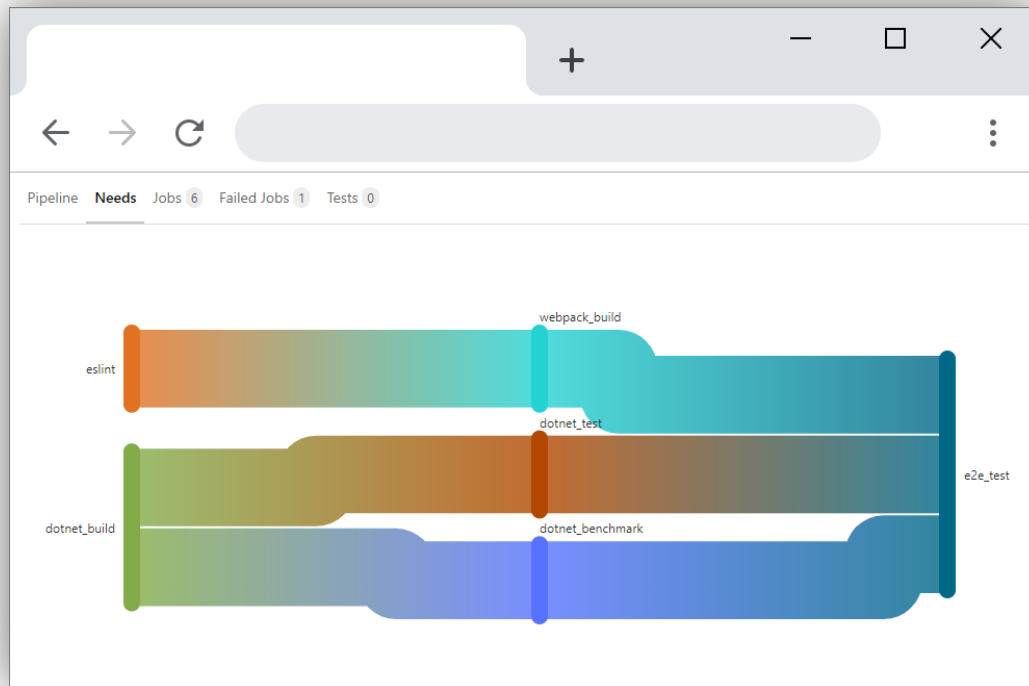


Figure A.2: Visualization of a pipeline on GitLab as *directed acyclic graph* (DAG); only available when utilizing *needs*.

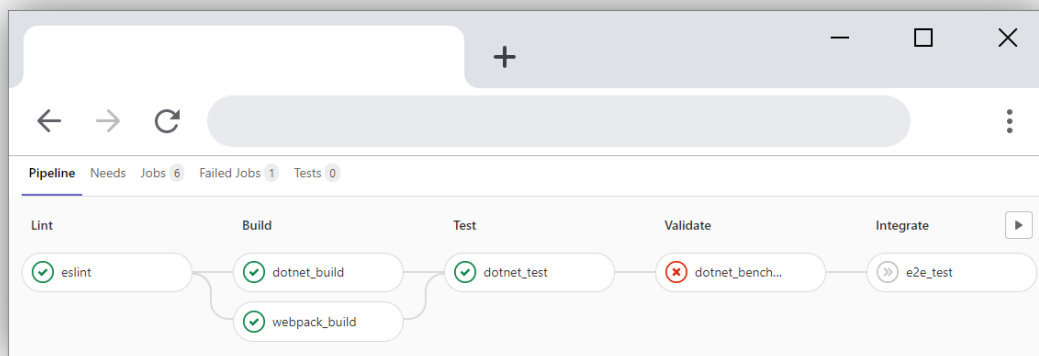


Figure A.3: Visualization of a pipeline on GitLab organized in stages

List of Figures

1.1	DevOps Infinity Wheel, as shown in [2].	1
2.1	<i>Shader Nodes</i> in Blender [5].	6
2.2	Unreal Engine Level Blueprint [16].	7
2.3	Data Transformation Pipeline in Azure Data Factory [35].	8
2.4	SemaphoreCI workflow for a mobile app; a template provided by the platform.	10
2.5	Pipeline visualization by GitLab, as shown in [31]	11
2.6	Mockup of a Visual Pipeline Editor, as shown in [45]	12
2.7	Deployment model for a modern web application, as shown in [46] (Fig. 2) .	12
3.1	The complete palette of the proposed DSL	18
4.1	Informal sketch of the studied transformation rules	27
4.2	Composition of the property unzipping (1) and property merging (2) rules. .	29
5.1	A cloud-based web application deployed via Docker, using Aurelia as SPA for the frontend that consumes a web API backend implemented via Quarkus.	32
6.1	Invalid configuration; leads to failing pipeline when creating or pushing a tag.	36
6.2	Invalid configuration; failing pipeline due to incorrect dependencies when creating or pushing a tag.	37
6.3	Mockup of visual authoring for conditions	42
6.4	Mockup of the condition embedded in the CI/CD workflow. Clicking the condition will enable the user to author the condition as shown in Figure 6.3	43
6.5	Visualization of a regular expression using Regulex	45
6.6	Build process of Rig, modelled in the DSL.	50

7.1	A cloud-based web application deployed via Docker, using Aurelia as SPA for the frontend that consumes a web API backend implemented via Quarkus. Modeled here with SemaphoreCI. Compare with Figure 5.1.	54
A.1	Example workflow of a cloud-based web application in SemaphoreCI	61
A.2	Visualization of a pipeline on GitLab as <i>directed acyclic graph</i> (DAG); only available when utilizing needs	62
A.3	Visualization of a pipeline on GitLab organized in stages	62

Bibliography

- [1] APPVEYOR SYSTEMS INC.: *Build configuration*. <https://www.appveyor.com/docs/build-configuration/#appveyoryml-and-ui-coexistence>. [Accessed 09-02-2021, Archived].
- [2] ATLASSIAN. PTY LTD: *What is DevOps?* <https://www.atlassian.com/devops>. [Accessed 09-02-2021, Archived].
- [3] BAKKI, AICHA, LAHCEN OUBAHSSI, SÉBASTIEN GEORGE and CHIHAB CHERKAOUI: *MOOCAT: A visual authoring tool in the cMOOC context*. *Educ. Inf. Technol.*, 24(2):1185–1209, Mar 2019.
- [4] BEN-KIKI, OREN, CLARK EVANS and INGY DÖT NET: *YAML Ain't Markup Language (YAML™) Version 1.2*. <https://yaml.org/spec/1.2/spec.html>, 2009. [Accessed 09-02-2021, Archived].
- [5] BLENDER CONTRIBUTORS: *Import & Export of Node Shaders – Blender Manual*. https://docs.blender.org/manual/en/latest/addons/import_export/node_shaders_info.html. [Accessed 18-03-2021, Archived].
- [6] BOOTSTRAP TEAM and CONTRIBUTORS: *Alerts · Bootstrap*. <https://getbootstrap.com/docs/4.0/components/alerts/>. [Accessed 20-03-2021, Archived].
- [7] BUSCH, DANIEL: *Cinco Version 2.0 Released*. <https://cinco.scce.info/cinco-release-2.0/>. [Accessed 25-03-2021].
- [8] CINCO DEV. TEAM: *Cinco Product Specification*. <https://gitlab.com/scce/cinco/-/wikis/Cinco-Product-Specification>. [Accessed 18-03-2021].
- [9] CINCO DEV. TEAM: *CINCO SCCE Meta Tooling Framework*. <https://cinco.scce.info/>. [Accessed 18-03-2021, Archived].
- [10] CINCO DEV. TEAM: *Meta Graph Language*. <https://gitlab.com/scce/cinco/-/wikis/Meta-Graph-Language>. [Accessed 18-03-2021].

- [11] CINCO DEV. TEAM: *Meta Style Language*. <https://gitlab.com/scce/cinco/-/wikis/Meta-Style-Language>. [Accessed 18-03-2021].
- [12] CIRCLE INTERNET SERVICES, INC.: *Setting up CircleCI*. <https://circleci.com/docs/2.0/getting-started/#setting-up-circleci>. [Accessed 09-02-2021, Archived].
- [13] DANIELSON, STEVE, USER "V-HEARYA", MIKE JACOBS, JULIA KULLA-MADER, NIKHIL DESAI, DAVID COULTER, THEANO PETERSEN, KRAIG BROCKSCHMIDT, USER "CHCOMLEY", USER "RAIYAN", SHANNON LEAVITT, GEORGE VERGHESE and USER "VIJAYMATEST1": *Create your first pipeline*. <https://docs.microsoft.com/en-us/azure/devops/pipelines/create-first-pipeline>. [Accessed 09-02-2021, Archived].
- [14] ECLIPSE FOUNDATION, INC. CONTRIBUTORS: *Rich Client Platform*. https://wiki.eclipse.org/Rich_Client_Platform. [Accessed 25-03-2021, Archived].
- [15] EPIC GAMES, INC.: *Blueprints Visual Scripting*. <https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/index.html>. [Accessed 09-02-2021, Archived].
- [16] EPIC GAMES, INC.: *Opening Doors | Unreal Engine Documentation*. <https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Timelines/Examples/OpeningDoors/index.html>. [Accessed 09-02-2021, Archived].
- [17] FAUTH, DIRK: *Building a “headless RCP” application with Tycho*. <http://blog.vogella.com/2020/01/20/building-a-headless-rcp-application-with-tycho/>. [Accessed 22-03-2021, Archived].
- [18] FOWLER, MARTIN: *Continuous Integration*. <https://martinfowler.com/articles/continuousIntegration.html>, 2006. [Accessed 09-02-2021, Archived].
- [19] FOWLER, MARTIN and MATTHEW FOEMMEL: *Continuous Integration*. <https://web.archive.org/web/20001201200300/https://martinfowler.com/articles/continuousIntegration.html>, 2000. [Accessed 09-02-2021].
- [20] GAVRILOVA, TATIANA, NATALIA STASH and RODOLPHE TEXIER: *Visual Authoring Tool for Web-Based Training*. IFAC Proceedings Volumes, 34(16):457 – 461, 2001. 8th IFAC Symposium on Analysis, Design and Evaluation of Human-Machine Systems (HMS 2001), Kassel, Germany, 18-20 September, 12001.
- [21] GITHUB, INC. and CONTRIBUTORS: *Hosting your own runners - GitHub Docs*. <https://docs.github.com/en/actions/hosting-your-own-runners>. [Accessed 19-03-2021, Archived].

- [22] GITHUB, INC. and CONTRIBUTORS: *Starter Workflows*. <https://github.com/actions/starter-workflows>. [Accessed 22-03-2021].
- [23] GITHUB, INC. and CONTRIBUTORS: *Using GitHub-hosted runners - GitHub Docs*. <https://docs.github.com/en/actions/using-github-hosted-runners>. [Accessed 19-03-2021, Archived].
- [24] GITHUB, INC. and CONTRIBUTORS: *Workflow syntax for GitHub Actions*. <https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions>. [Accessed 20-03-2021].
- [25] GITLAB INC. and CONTRIBUTORS: *GitLab CI/CD Examples*. <https://docs.gitlab.com/ee/ci/examples/#cicd-templates>. [Accessed 22-03-2021].
- [26] GITLAB INC. and CONTRIBUTORS: *GitLab Runner / GitLab*. <https://docs.gitlab.com/runner/>. [Accessed 19-03-2021, Archived].
- [27] GITLAB INC. and CONTRIBUTORS: *Keyword reference for the .gitlab-ci.yml file*. <https://docs.gitlab.com/ee/ci/yaml/README.html>. [Accessed 20-03-2021].
- [28] GITLAB INC. and CONTRIBUTORS: *Pipeline schedules*. <https://docs.gitlab.com/ee/ci/pipelines/schedules.html>. [Accessed 27-03-2021, Archived].
- [29] GITLAB INC. and CONTRIBUTORS: *Using GitLab CI/CD with a GitHub repository*. https://docs.gitlab.com/ee/ci/ci_cd_for_external_repos/github_integration.html. [Accessed 26-03-2021, Archived].
- [30] GITLAB INC. and CONTRIBUTORS: *Validate .gitlab-ci.yml syntax with the CI Lint tool*. <https://docs.gitlab.com/ee/ci/lint.html>. [Accessed 09-02-2021, Archived].
- [31] GITLAB INC. AND CONTRIBUTORS: *Pipeline Editor*. https://docs.gitlab.com/ee/ci/pipeline_editor/index.html. [Accessed 09-02-2021, Archived].
- [32] HERSHKOVITCH, DOV, GITLAB INC. and CONTRIBUTORS: *Visual Builder for Pipeline Editor*. <https://gitlab.com/groups/gitlab-org/-/epics/4499>. [Accessed 20-03-2021].
- [33] KOPETZKI, D., M. LYBECAIT, S. NAUJOKAT and B. STEFFEN: *Towards Language-to-Language Transformation*. Int. J. on Software Tools for Technology Transfer (STTT), 2020.
- [34] MICROSOFT CORPORATION: *Azure Data Factory documentation*. <https://docs.microsoft.com/en-us/azure/data-factory/>. [Accessed 18-03-2021, Archived].
- [35] MICROSOFT CORPORATION: *Azure Data Factory documentation*. <https://docs.microsoft.com/en-us/azure/data-factory/>. [Accessed 18-03-2021, Archived].

- [36] MICROSOFT CORPORATION and CONTRIBUTORS: *YAML schema reference*. <https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema#syntax-highlighting>. [Accessed 09-02-2021, Archived].
- [37] OBJECT MANAGEMENT GROUP (OMG): *OMG® Unified Modeling Language® (OMG UML®) Version 2.5.1*. OMG Document Number: formal/2017-12-05 (<https://www.omg.org/spec/UML/2.5.1/PDF>), 2017.
- [38] PUNDSACK, MARK, GITLAB INC. and CONTRIBUTORS: *Out-of-sequence job execution using directed acyclic graphs (DAG) MVC*. <https://gitlab.com/gitlab-org/gitlab-foss/-/issues/47063>. [Accessed 24-03-2021].
- [39] REDHAT, INC. and CONTRIBUTORS: *Quarkus - Building applications with Maven*. <https://quarkus.io/guides/maven-tooling>. [Accessed 22-03-2021, Archived].
- [40] RENDERED TEXT: *Artifacts*. <https://docs.semaphoreci.com/essentials/artifacts/>. [Accessed 20-03-2021, Archived].
- [41] RENDERED TEXT: *Creating Your First Project*. <https://docs.semaphoreci.com/guided-tour/creating-your-first-project/>. [Accessed 22-03-2021, Archived].
- [42] RENDERED TEXT: *Deploying With Promotions*. <https://docs.semaphoreci.com/guided-tour/deploying-with-promotions/>. [Accessed 22-03-2021, Archived].
- [43] RENDERED TEXT: *Pipeline YAML Reference*. <https://docs.semaphoreci.com/reference/pipeline-yaml-reference/>. [Accessed 22-03-2021, Archived].
- [44] SINGH, MINANSHU, TOBIAS KOPPERS, GERARD O'NEILL, USER "ROUZBEH84" and GREG VENECH: *Hot Module Replacement*. <https://webpack.js.org/concepts/hot-module-replacement/>. [Accessed 22-03-2021, Archived].
- [45] SOTNIKOVA, NADIA, GITLAB INC. and CONTRIBUTORS: *3 Year Vision - Verify Pipeline Authoring*. <https://gitlab.com/groups/gitlab-org/-/epics/4534>. [Accessed 20-03-2021].
- [46] TEGELER, T., F. GOSSEN and B. STEFFEN: *A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications*. In *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, pages 1–6, 2019.
- [47] TRAVIS CI GMBH: *Travis CI Tutorial*. <https://docs.travis-ci.com/user/tutorial/>. [Accessed 09-02-2021, Archived].
- [48] USER "ELBOLETAIRE": *Use GitLab CI to run tests locally?* <https://stackoverflow.com/a/36358790/1360803>. [Accessed 09-02-2021].

- [49] USER "IVANAUR": *GitLab support in Semaphore 2.0*. <https://github.com/semaphoreci/semaphore/issues/30#issuecomment-525297766>. [Accessed 18-03-2021, Archived].
- [50] WHARTON, JAKE: *GSON Serialization/Deserialization of Java 8 Date API*. <https://github.com/google/gson/issues/1059#issuecomment-292623118>. [Accessed 20-03-2021, Archived].
- [51] WIKIPEDIA CONTRIBUTORS: *Jackson (API)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Jackson_\(API\)&oldid=999087269](https://en.wikipedia.org/w/index.php?title=Jackson_(API)&oldid=999087269), 2021. [Online; accessed 20-March-2021].
- [52] WIKIPEDIA CONTRIBUTORS: *Rigging* — *Wikipedia, The Free Encyclopedia*, 2021. [Online; accessed 20-March-2021].
- [53] WRIGHT, AUSTIN, HENRY ANDREWS, BEN HUTTON and GREG DENNIS: *JSON Schema: A Media Type for Describing JSON Documents*. <https://tools.ietf.org/html/draft-handrews-json-schema-02>. [Accessed 09-02-2021].